technische universität
dortmund

Masterarbeit

Konfiguration und Parameteroptimierung eines Evolutionären Algorithmus für die Maschinenbelegungsplanung mittels kombinatorischer Logik

Dominik Mäckel

Oktober 2023

Gutachterin 1:
Dr.-Ing. Christin Schumacher
Technische Universität Dortmund
Fakultät Maschinenbau
Institut für Transportlogistik
https://itl.mb.tu-dortmund.de/

Gutachter 2:
Prof. Dr.-Ing. Jakob Rehof
Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Softwareengineering (LS14)
https://se.cs.tu-dortmund.de/

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Tabellenverzeichnis iv

Algorithmenverzeichnis v

Abkürzungsverzeichnis vi

Symbolverzeichnis vii

1 Einleitung 1

2 Grundlagen 3

2.1 Maschinenbelegungsplanung . . . 3

2.1.1 Notation . . . 4

2.1.2 Maschinenumgebung α . . . 4

2.1.3 Produktionscharakteristika β . . . 6

2.1.4 Zielfunktion γ . . . 7

2.1.5 Einordnung in den Kontext der Arbeit . . . 7

2.2 Evolutionäre Algorithmen . . . 9

2.2.1 Aufbau . . . 10

2.2.2 Eltern- und Überlebensselektion . . . 11

2.2.3 Repräsentation . . . 15

2.2.4 Mutation . . . 16

2.2.5 Rekombination . . . 18

2.3 Parameteroptimierung . . . 19

2.3.1 Gittersuche . . . 20

2.3.2 Modellbasierte Parameteroptimierung . . . 21

2.4 Kombinatorische Logik und CLS . . . 22

3 Stand der Forschung 24

3.1 Evolutionäre Algorithmen . . . 24

3.2 Parameteroptimierung . . . 25

3.3 CLS . . . 26

3.4 Benchmarks . . . 26

4 Implementierung des Syntheseframeworks 28

4.1 Repräsentation der Lösung . . . 28

4.2 Komponentisierung des Evolutionären Algorithmus . . . 31

4.2.1 Generatoren für die Initiallösungen . . . 32

4.2.2 Eltern- und Überlebensselektion . . . 33

4.2.3 Rekombination . . . 34
4.2.4 Mutation . . . 36
4.2.5 Stufenstrategie . . . 38
4.2.6 Framework . . . 39
4.2.7 Verknüpfung von Algorithmen und Kombinatoren . . . 41
4.3 Dynamische Komponentensuche . . . 46
4.3.1 Modulsuche . . . 47
4.3.2 Erstellung eines Modulmappings . . . 47
4.3.3 Identifizierung der Voraussetzungen des Zielkombinators . . . 50
4.3.4 Ausführung der Parameterkonfigurationen . . . 51
4.4 Mathematisches Optimierungsmodell . . . 52
4.4.1 Definition der Parameter und Variablen . . . 52
4.4.2 Formulierung des ILP . . . 53

5 Evaluation **57**
5.1 Benchmarks . . . 57
5.2 Vergleich der Lösungen . . . 58
5.3 Gittersuche . . . 59
5.3.1 Lösungsgeneratoren . . . 61
5.3.2 Selektionsoperatoren . . . 62
5.3.3 Genetische Operatoren . . . 63
5.3.4 Populationsgröße und Elternschaft . . . 66
5.4 Modellbasierte Parameteroptimierung . . . 67

6 Diskussion der Ergebnisse **71**

7 Zusammenfassung und Ausblick **73**

Literatur **75**

Anhang **I**

Abbildungsverzeichnis

1 Komplexitätshierarchie der Maschinenumgebungen. . . . 4
2 Gantt-Diagramm eines Ablaufplanes mit eingezeichnetem Makespan. . . . 7
3 Schematische Darstellung eines beispielhaften hybriden Flow Shops. . . . 8
4 Schematischer Aufbau eines Evolutionären Algorithmus. . . . 10
5 Repräsentation eines Individuums für Permutationsprobleme. . . . 16
6 2-Swap Mutation eines Individuums. . . . 16
7 Translokations-Mutation eines Individuums. . . . 17
8 Order-Based-Crossover. . . . 18
9 FCL($\cap, \leq$). . . . 23
10 Darstellung eines Ausschnitts des Schedule-Objekts in UML. . . . 29
11 Darstellung des Allocation-Objekts in UML. . . . 30
12 Beispiel für die Darstellung der Maschinenzuweisungen. . . . 30
13 Übersicht über die implementierten Operatoren des Evolutionären Algorithmus. . . . 32
14 Semantischer Typ des Zielkombinators. . . . 42
15 Kombinator für den Basisalgorithmus. . . . 43
16 Schematische Vorgehensweise zur automatisierten Komponentensuche. . . . 46
17 Sequenzdiagramm der Erstellung des Modulmappings. . . . 48
18 Vergleich der Lösungsgeneratoren vor und nach der Optimierung. . . . 61
19 Vergleich der Selektionsoperatoren. . . . 62
20 Isolierter Vergleich der Crossover- und Mutationsoperatoren. . . . 63
21 GG-Plot der Beziehungen zwischen Crossover- und Mutationsoperatoren und -wahrscheinlichkeiten. . . . 64
22 Vergleich der Crossover- und Mutationswahrscheinlichkeiten. . . . 65
23 Isolierter Vergleich der Populationsgröße und Elternschaft. . . . 66
24 Ablauf der gesamten Parameteroptimierung. . . . 67
25 Konvergenzgraph der modellbasierten Optimierung. . . . 70
26 Darstellung aller Felder des Schedule-Objekts in UML. . . . I
27 Darstellung aller öffentlicher Methoden des Schedule-Objekts in UML. . . . II
28 GG-Plot der Beziehungen zwischen Populationsgröße und Elternschaft. . . . VII

Tabellenverzeichnis

1 Verwendete Maschinenumgebungen der Maschinenbelegungsplanung in dieser Arbeit nach Pinedo und Ruiz et al. 5
2 Verwendete Nebenbedingungen der Maschinenbelegungsplanung in dieser Arbeit nach Pinedo und Ruiz et al. 6
3 Benchmarks für hybride Flow Shops aus der Literatur nach Kordus. 27
4 Übersicht der geladenen Module bei der Komponentensuche. 47
5 Übersicht des Modulmappings bei der Komponentensuche. 49
6 Übersicht der gefundenen numerischen Parameter und Inhabitanten für die Voraussetzungen des Zielkombinators. 50
7 Entscheidungsvariablen des ILP. 52
8 Variablen des ILP. 53
9 Konfigurationen der Benchmarks. 57
10 Genutzte Hardware für die Optimierungsläufe des ILP und der Parameteroptimierung. 58
11 Ergebnisse der exakten Optimierung. 58
12 Gewählte Parameter für die Gittersuche. 60
13 Ergebnisse der Gittersuche. 60
14 Konfiguration des Parameteroptimierers. 68
15 Ergebnisse der modellbasierten Optimierung. 69

Algorithmenverzeichnis

1	Selektionsoperator zufällige Wahl mit fester Anzahl aus der Literatur.	12
2	Selektionsoperator zufällige Wahl mit stochastischer Anzahl aus der Literatur.	12
3	Elitistisches Framework für Selektionsoperatoren aus der Literatur.	13
4	(μ+,λ) Selektion aus der Literatur.	14
5	Selektionsoperator Turnierselektion aus der Literatur.	14
6	Mutationsoperation 2-Swap aus der Literatur.	16
7	Mutationsoperation Translokation aus der Literatur.	17
8	Order-Based-Crossover aus der Literatur.	19
9	Lösungsgenerator zufällige Verteilung.	32
10	Lösungsgenerator ausgeglichene Verteilung.	33
11	Selektionsoperator zufällige Wahl.	33
12	Selektionsoperator elitistische Wahl.	33
13	Selektionsoperator Turnierselektion.	34
14	Rekombinationsoperator Order-Based.	35
15	Rekombinationsoperator Maschinenzuweisungen.	36
16	Mutationsoperator Shift.	37
17	Mutationsoperator Swap.	37
18	Stufenstrategie Earliest Completion Time (ECT).	38
19	Basisalgorithmus.	40
20	Klasse MutationShiftCombinator.	44
21	run-Methode der Klasse EvolutionBaseCombinator.	45
22	Beispiel für einen Parametervektor.	51
23	Rekombinationsoperator Order-Based.	III
24	Rekombinationsoperator Maschinenzuweisungen.	IV
25	Mutationsoperator Shift.	V
26	Mutationsoperator Swap.	VI

Abkürzungsverzeichnis

CLS Combinatory Logic Synthesizer.

PI Aquisitionsfunktion *Probability of Improvement*.

EI Aquisitionsfunktion *Expected Improvement*.

LCB Aquisitionsfunktion *Lower Confidence Bound*.

FCL Endliche kombinatorische Logik.

NSGA-II Non-dominated-sorting genetic algorithm II.

UML Unified Modeling Language.

EST Earliest Start Time.

ECT Earliest Completion Time.

lb Lower Bound.

ub Upper Bound.

RPD Relative Percentage Deviation.

DoE Design of Experiments.

OBX Crossoveroperator Order Based.

MA Crossoveroperator Machine Allocation.

Symbolverzeichnis

$\mathbb{N}$ Natürliche Zahlen ohne null.

$\mathbb{N}_0$ Natürliche Zahlen mit null.

$\mathbb{P}_n$ Permutationsraum mit n Elementen.

$\mathcal{NP}$ Komplexitätsklasse nichtdeterministisch in Polynomialzeit berechenbar.

α Maschinenumgebung.

β Nebenbedingungen.

γ Zielfunktion.

N Menge der Aufträge.

M Menge der Bearbeitungsstufen.

M_i Menge der Maschinen auf Bearbeitungsstufe i.

n Anzahl der Aufträge.

m Anzahl der Bearbeitungsstufen.

m_i Anzahl der Maschinen auf Bearbeitungsstufe i.

j, k Aufträge $j, k \in 1 \ldots n$.

i Bearbeitungsstufe $i \in 1 \ldots m$.

l Maschine $l \in 1 \ldots m_i$.

1 Maschinenumgebung Einmaschinenproblem.

PM Maschinenumgebung Parallele identische Maschinen.

QM Maschinenumgebung Parallele uniforme Maschinen.

RM Maschinenumgebung Parallele unabhängige Maschinen.

Fm Maschinenumgebung Flow Shop.

FHm Maschinenumgebung hybrider Flow Shop.

p_{ilj} Prozesszeit von Auftrag j auf Maschine l auf Bearbeitungsstufe i.

v_{il} Geschwindigkeit von Maschine l auf Bearbeitungsstufe i.

$skip$ Nebenbedingung Stufenauslassung.

M_j Nebenbedingung Maschinenqualifikationen.

r_m Nebenbedingung Maschinenstartzeiten.

lag Nebenbedingung Stufenwechselzeit.

S_{iljk} Nebenbedingung Reihenfolgenabhängige Umrüstzeiten.

A_{iljk} Nebenbedingung Antizipative Umrüstzeiten.

$fmls$ Nebenbedingung Auftragsfamilien.

$prec$ Nebenbedingung Reihenfolgenbedingung.

$batch$ Nebenbedingung Stapelverarbeitung.

ddw Nebenbedingung Fälligkeitszeitfenster.

T_j Verspätung von Auftrag j.

E_j Verfrühung von Auftrag j.

w_j Gewichtung w von Auftrag j.

V Große Zahl für das ILP.

X_{iljk} Entscheidungsvariable für die Auftragsketten des ILP.

C_{ij} Vertigstellungszeitpunkt von Auftrag j auf Bearbeitungsstufe i.

C_{max} Zielfunktion Makespan.

$\sum T_j$ Zielfunktion Total tardiness.

$\sum(w'_j E_j^{dw} + w_j T_j^{dw})$ Zielfunktion Weighted total earliness and tardiness.

μ Populationsgröße.

λ Elternschaftsgröße.

p_c Crossoverwahrscheinlichkeit.

p_m Mutationswahrscheinlichkeit.

pop_size Populationsgröße.

par_size Elternschaftsgröße.

n_{sel} Anzahl gewählter Individuen bei der Selektion.

p_{sel} Wahrscheinlichkeit der Auswahl bei der Selektion.

κ Gruppengröße der Turnierselektion, Parameter der Aquisitionsfunktion LCB.

π Chromosom.

c_1, c_2 Crossoverpunkte.

ψ Parametervektor.

Γ CLS-Repository.

τ Zieltyp der Inhabitation.

1 Einleitung

In einer Produktionsumgebung ist eine gute Maschinenbelegungsplanung Schlüssel zur effizienten Nutzung der zur Verfügung stehenden Ressourcen und wirtschaftlichem Handeln. Durch den Einfluss diverser Produktionscharakteristika können häufig keine bestehenden Lösungsalgorithmen zum Einsatz kommen. Die Entwicklung angepasster Algorithmen für ein spezielles Szenario ist zeitaufwändig und kostenintensiv. Gupta [1] zeigte, dass bereits ein paralleles Maschinenproblem mit zwei identischen Maschinen $\mathcal{NP}$-schwer ist. Folglich ist es für die meisten realistischen Maschinenbelegungsprobleme nicht effizient möglich, eine optimale Lösung zu generieren. Stattdessen werden häufig Heuristiken eingesetzt, um mit geringem Rechenaufwand eine möglichst gute Lösung zu berechnen. Bei der Auswahl von Algorithmen ergeben sich zunächst die folgenden zwei Fragestellungen:

1. Welches Verfahren bzw. welche Vorgehensweise wird gewählt?
2. Wie wird der gewählte Algorithmus parametrisiert?

Zur automatisierten Auswahl von Heuristiken auf Basis von Problemspezifikationen existieren bereits Vorarbeiten aus der vorangegangenen Bachelorarbeit [2, 3] und der Projektgruppe Autoschedule [4]. Hier wurde der *Combinatory Logic Synthesizer (CLS)* [5] eingesetzt, um aus einer Menge von Algorithmenkomponenten anwendbare zu filtern und zu lauffähigen Instanzen zusammenzufügen. Somit ist es möglich, nur einzelne Komponenten bekannter Heuristiken an spezielle Produktionscharakteristika anzupassen, anstatt sie vollständig neu designen zu müssen. Durch die Wiederverwendung dieser Komponenten können diese Modifikationen automatisiert von mehreren Lösungsverfahren verwendet werden, was den Anpassungsaufwand verringert und die Menge der zur Verfügung stehenden Verfahren erhöht.

Ziel dieser Masterarbeit ist die Weiterentwicklung dieser Vorarbeiten. Konkret soll die zweite Forschungsfrage beantwortet werden, indem ein Vorgehensmodell zur automatischen Parametrisierung von Algorithmen entworfen und implementiert wird. Am Beispiel eines Evolutionären Algorithmus soll gezeigt werden, wie dieser automatisiert sowohl konfiguriert als auch parametrisiert werden kann. Algorithmenparameter lassen sich dabei in zwei Kategorien unterteilen [6]:

1. Numerische Parameter, die reellwertig oder ganzzahlig und ggf. nach oben und unten beschränkt sein können. Beispiele sind im Umfeld Evolutionärer Algorithmen die Populationsgröße $\mu \in \mathbb{N}$ oder die Mutationswahrscheinlichkeit $p_m \in [0, 1]$. Ziel der Suche ist es, den optimalen Wert im gültigen Intervall zu finden.
2. Kategorische Parameter, wie die Wahl des Selektionsverfahrens aus einer gegebenen finiten Menge an Algorithmen. Zwischen den Elementen kann im Gegensatz zu numerischen Parametern kein Distanzmaß angenommen werden.

Das zu implementierende Framework bildet eine Erweiterung des bestehenden Konzeptes zur Algorithmenkomponentisierung und anschließenden Synthese von Maschinenbelegungsalgorithmen mittels kombinatorische Logik. In dieser Arbeit soll gezeigt werden, dass diese auch zur Steuerung der kategorischen Parameter des Evolutionären Algorithmus eingesetzt werden kann. Zusätzlich sollen auch die numerischen Parameter einer Komposition über das Framework automatisiert konfigurierbar sein. Eine Erweiterbarkeit des Frameworks um neue Algorithmen soll durch die automatisierte Suche nach verfügbaren Kombinatoren für die Inhabitation im ersten Syntheseschritt sichergestellt werden. Auch die Identifikation von Variationspunkten und numerischen Parametern soll automatisiert erfolgen, um die Integration neuer parameterbehafteter Verfahren zu erleichtern.

Ein weiteres Ziel der Arbeit ist die modulare Anbindung von Parameteroptimierern an das Syntheseframework, sodass diese unabhängig vom entwickelten Vorgehensmodell zur Komposition der Experimente bleiben und somit austauschbar sind. Es sollen zwei Verfahren zur automatisierten Parameteroptimierung eingesetzt und verglichen werden, um geeignete Komponenten und Parameter für ein gegebenes Testproblem zu identifizieren. Dazu kann zunächst eine Gittersuche implementiert werden, die den Parameterraum systematisch durchsucht. Dies liefert eine erste Einordnung der Parameter und ermöglicht durch deren äquidistante Wahl einen direkten paarweisen Vergleich. Darüber hinaus können durch die systematische Wahl die Beziehungen zwischen mehreren Einflussfaktoren untersucht werden. Anschließend an die Gittersuche soll eine modellbasierte Parameteroptimierung durchgeführt werden. Diese nutzt ein Surrogatmodell der Zielfunktion zur Prädiktion des nächsten Versuchspunktes, was die Anzahl der Experimente auf Kosten der Modellbildung reduziert. Alternative Parameteroptimierungsverfahren werden in Abschnitt 3 besprochen.

Zusammengefasst soll die Arbeit die Möglichkeit einer automatisierten Parameteroptimierung eines aus Kombinatoren synthetisierbaren Algorithmus mittels modular angebundenen Parameteroptimierungsverfahren aufzeigen und evaluieren. Der Aufbau der Arbeit lässt sich wie folgt zusammenfassen:

In Abschnitt 2 werden zunächst die Grundlagen zur Maschinenbelegungsplanung, zu Evolutionären Algorithmen, zur Parameteroptimierung und zu kombinatorischer Logik, die in der Arbeit Anwendung finden, dargelegt. In Abschnitt 3 wird anschließend auf die aktuelle Forschung in den besprochenen Themengebieten eingegangen. Die Implementierung des Syntheseframeworks wird in Abschnitt 4 erläutert. Dabei wird zunächst auf die implementierten Komponenten des Evolutionären Algorithmus und anschließend auf die Konzeption der Parameteroptimierung eingegangen. In Abschnitt 5 erfolgt die Anwendung der Parameteroptimierungsverfahren auf ein Testproblem und eine Evaluation der ermittelten Parameter. Die Arbeit schließt mit einer Diskussion der Ergebnisse in Abschnitt 6 und einer Zusammenfassung in Abschnitt 7.

2 Grundlagen

In diesem Abschnitt der Arbeit werden zunächst die Grundlagen der verwendeten Konzepte erläutert. Im ersten Teil werden Maschinenbelegungsprobleme, ihre Klassifikation und Notation als Anwendungsfall der Arbeit beschrieben. Anschließend wird eine Übersicht über die in der Literatur gängigen algorithmischen Komponenten des zu implementierenden Evolutionären Algorithmus gegeben, der zur Lösung eines Maschinenbelegungsproblems verwendet werden soll. Nach der Betrachtung der beiden verwendeten Parameteroptimierungsverfahren zur Parametrisierung des Evolutionären Algorithmus schließt der Abschnitt mit der Erläuterung der Typtheoretischen Grundlagen des dazu verwendeten Syntheseframeworks CLS.

2.1 Maschinenbelegungsplanung

Die Maschinenbelegungsplanung ist eine Disziplin der Informatik und Logistik, die sich der effizienten Zuweisung von Arbeitsaufträgen (*engl. „jobs“*) auf verfügbare Maschinen widmet. Sie ist essenziell für die Planung und Steuerung von Fertigungsprozessen in der Industrie, die Verteilung von Aufgaben auf Mitarbeitende, die Zuweisung von Rechenzeit auf parallel rechnender Hardware und viele weitere vergleichbare Aufgaben der Ressourcenverteilung [7, S. 3]. Insbesondere kann eine effizientere Planung von beispielsweise Fertigungsprozessen zu einer Steigerung der Produktivität bzw. Reduzierung der Produktionskosten und damit zu erhöhter Wettbewerbsfähigkeit führen. Sie ist somit insbesondere ein wichtiges Instrument zur Optimierung der Wirtschaftlichkeit von Fertigungsprozessen [8, S. 1 f.][9, S. 1].

Die Maschinenbelegungsplanung wird relevant, sobald mehrere Aufträge konkurrierend auf dieselbe Ressource zugreifen und eine Entscheidung bezüglich der Priorisierung und somit Reihenfolge erfolgt. Konzepte der Maschinenbelegungsplanung werden in der Literatur häufig aus Sicht einer Produktion beschrieben, indem von „Aufträgen“ und „Maschinen“ gesprochen wird [7, S. 3]. Dies schränkt jedoch in keiner Weise die Anwendbarkeit der Konzepte auf andere Probleme zum Beispiel aus der Informatik, Kommunikationstechnik oder der Logistik ein.

In der Maschinenbelegungsplanung kommt es durch die Heterogenität der produktionsspezifischen Anforderungen an die Planungssysteme zu einer großen Bandbreite an Problemspezifikationen. Um überhaupt Lösungsansätze für bestehende Planungsprobleme entwickeln zu können, werden die Prozesse der Fertigung abstrahiert. Das Abstraktionslevel bestimmt dabei maßgeblich die Passgenauigkeit der entwickelten Lösungsmethoden, steuert aber auch die Entwicklungs- und Problemkomplexität, die sich letztlich in der Entwicklungszeit und auch der resultierenden Laufzeit des Planungssystems widerspiegelt [10, S. 11 f.].

Ziel der Maschinenbelegungsplanung ist die Erzeugung eines gültigen und gemäß der Zielfunktion möglichst guten Ablaufplanes. Dieser enthält Informationen über die Reihenfolge der Aufträge (*sequencing*), ihrer Maschinenzuweisungen (*machine allocation*) und der Start- und Endzeiten (*timing*) auf den entsprechenden Maschinen [11, S. 2]. Ein Ablaufplan ist gültig, wenn er alle Aufträge mit den entsprechenden Prozesszeiten überlappungsfrei eingeplant hat und gegen keine der Nebenbedingungen verstößt.

2.1.1 Notation

Um Maschinenbelegungsprobleme kategorisieren zu können, schlugen Graham et al. [12] ein Notationsschema vor, das das zu modellierende Maschinenbelegungsproblem als 3-Tupel

$$\alpha \mid \beta \mid \gamma$$

ausdrückt. Die drei Felder α, β, γ beschreiben die verschiedenen Aspekte des Maschinenbelegungsproblems, die im Folgenden einzeln erläutert werden. Dazu wird in dieser Arbeit konsistent die erweiterte Notation von Ruiz et al. [13] genutzt. Weiterhin wird die Menge der Aufträge $j \in 1 \dots n$ (bzw. k falls zwei verschiedene Aufträge existieren) als N bezeichnet. Die Menge der Bearbeitungsstufen $i \in 1 \dots m$ wird als M und die Menge der Maschinen mit $l \in 1 \dots m_i$ als M_i notiert [9, S. 13][13, S. 3].

2.1.2 Maschinenumgebung α

Die Maschinenumgebung α definiert die Anzahl und Anordnung der verfügbaren Maschinen des Maschinenbelegungsproblems. Dies können eine einzelne Maschine bis hin zu mehreren parallelen Produktionsstraßen oder individuellen Arbeitsstationen sein. Die typischen Maschinenumgebungen bis zur Klasse der hybriden Flow Shops, die in dieser Arbeit verwendet wird, sind in Tabelle 1 angegeben. Bereits ohne Nebenbedingungen sind die angegebenen Maschinenbelegungsprobleme aus komplexitätstheoretischer Sicht ab zwei parallelen Maschinen $\mathcal{NP}$-schwer [1, S. 1 f.]. Die Komplexitätshierarchie der angegebenen Problemklassen ist in Abbildung 1 gezeigt.

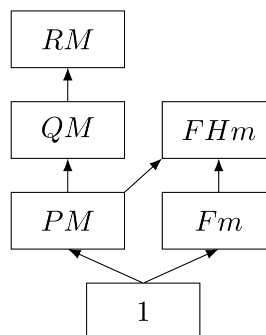


Abbildung 1: Komplexitätshierarchie der Maschinenumgebungen nach Pinedo [9, S. 27].

α	Maschinenumgebung
1	**Einmaschinenproblem:** Es steht genau eine Maschine zur Verfügung, auf der alle Aufträge bearbeitet werden müssen.
PM	**Parallele identische Maschinen:** m parallele identische Maschinen, wobei jeder Auftrag von genau einer der Maschinen bearbeitet werden muss. Die Wahl der Maschine ist irrelevant, da sie identisch sind. $P1$ ist äquivalent zum Einmaschinenproblem.
QM	**Parallele uniforme Maschinen:** m parallele Maschinen mit unterschiedlichen Geschwindigkeiten v_l. die Prozesszeit eines Auftrages j ergibt sich auf Maschine l als $p_{lj} = p_j/v_l$.
RM	**Parallele unabhängige Maschinen:** m parallele Maschinen, wobei jeder Auftrag für jede Maschine unterschiedliche Prozesszeiten haben kann.
Fm	**Flow Shop:** m Maschinen in Serie. Jeder Auftrag muss auf allen Maschinen in derselben vorgegebenen Reihenfolge bearbeitet werden.
FHm	**Hybrider Flow Shop:** m Bearbeitungsstufen. Jeder Auftrag muss auf allen Bearbeitungsstufen jeweils auf genau einer Maschine bearbeitet werden. Pro Bearbeitungsstufe stehen ggf. mehrere parallele Maschinen zur Verfügung. Diese können entsprechend wieder als PM, QM oder RM kategorisiert werden. Ein hybrider Flow Shop mit zwei Bearbeitungsstufen mit vier identischen Maschinen auf der ersten und zwei unabhängigen Maschinen auf der zweiten Stufe kann also im α-Feld der Graham-Notation in der Notationsform nach Ruiz et al. [13] als $FH2, (P4^1, R2^2)$ dargestellt werden.

Tabelle 1: Verwendete Maschinenumgebungen der Maschinenbelegungsplanung in dieser Arbeit nach Pinedo [9] und Ruiz et al. [13].

2.1.3 Produktionscharakteristika β

Das Feld β notiert eventuell vorhandene zusätzliche Produktionscharakteristika. Die in dieser Arbeit verwendeten sind in Tabelle 2 nach der Definition von Ruiz et al. [13] beschrieben, wobei nur *skip* und M_j bei der Implementierung des Syntheseframeworks bislang berücksichtigt wurden.

β	Nebenbedingung
$skip$	**Stufenauslassung:** Aufträge müssen nicht auf allen Stufen gefertigt werden.
M_j	**Maschinenqualifikationen:** Aufträge können nur auf den für sie qualifizierten Maschinen gefertigt werden. Für jeden Auftrag muss auf jeder Stufe, auf der er gefertigt werden muss, mindestens eine Maschine qualifiziert sein.
r_m	**Maschinenstartzeiten:** Maschinen können erst verspätet verfügbar sein. Für jede Maschine existiert eine Startzeit $r_m \geq 0$. Vor der Startzeit kann kein Auftrag begonnen werden.
lag	**Stufenwechselzeit:** Es existieren Transportzeiten zwischen den Bearbeitungsstufen.
S_{iljk}	**Reihenfolgenabhängige Umrüstzeiten:** Zwischen der Bearbeitung zweier Aufträge existiert eine Umrüstzeit abhängig von dem die Maschine verlassenden und dem nächsten Auftrag.
A_{iljk}	**Antizipative Umrüstzeiten:** Einige Umrüstzeiten können bereits vor Ankunft des Auftrages an der Maschine begonnen werden, sollte die Maschine frei sein. Nur in Kombination mit S_{iljk}.
$fmls$	**Auftragsfamilien:** Aufträge können zu Auftragsfamilien gruppiert werden. Aufträge derselben Familie benötigen keine Umrüstzeit zwischen den Bearbeitungen.
$prec$	**Reihenfolgenbedingung:** Es existiert eine Prioritätsreihenfolge der Aufträge. Niedrig priorisierte Aufträge können nicht bearbeitet werden, während höher priorisierte unbearbeitet sind.
$batch$	**Stapelverarbeitung:** Maschinen können mehrere Aufträge gleichzeitig bearbeiten.
ddw	**Fälligkeitszeitfenster:** Es existieren Fälligkeiten für die Fertigstellung von Aufträgen als Zeitfenster. Findet die Fertigstellung zu früh oder zu spät statt, führt dies zu einer Bestrafung in der Zielfunktion. Nur in Kombination mit Fälligkeitsbasierten Zielfunktionen.

Tabelle 2: Verwendete Nebenbedingungen der Maschinenbelegungsplanung in dieser Arbeit nach Pinedo [9] und Ruiz et al. [13].

2.1.4 Zielfunktion γ

Die Zielfunktion bestimmt die Bewertung eines generierten Ablaufplanes und ermöglicht einen qualitativen Vergleich zweier Lösungen während der Optimierung. Die am häufigsten verwendete Zielfunktion ist der *Makespan*

$$C_{max} = \max_{j \in N, i \in M} \{ \ C_{ij} \ \}$$

als Maximum der Fertigstellungszeitpunkte der Aufträge C_{ij} unter Annahme der Produktionsstarts bei $t = 0$ [14, S. 17 f.]. Der *Makespan* drückt also die Gesamtdauer für die Fertigstellung alle Aufträge aus. Ein Gantt-Diagramm eines Ablaufplanes für eine paralleles Maschinenproblem mit eingezeichnetem Makespan ist in Abbildung 2 gezeigt.

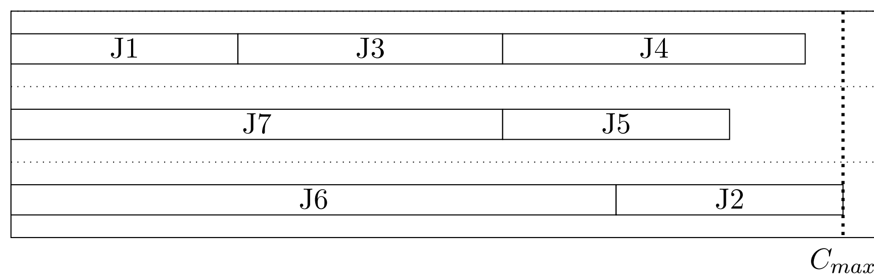


Abbildung 2: Gantt-Diagramm eines Ablaufplanes mit eingezeichnetem Makespan.

Für die Implementierung des Syntheseframeworks wird der *Makespan* zugrunde gelegt. Im Laufe der Arbeit tauchen zudem die Zielfunktionen *Total tardiness* $\sum T_j$ als Summe der Verspätungen T_j und die gewichtete Verfrühung und Verspätung $\sum(w'_j E_j^{dw} + w_j T_j^{dw})$ als Summe aller Verfrühungen E_j^{dw} und Verspätungen T_j^{dw} mit den Gewichten w_j und w'_j auf.

2.1.5 Einordnung in den Kontext der Arbeit

In dieser Arbeit wird ein hybrider Flow Shop modelliert. Dieser ist, wie bereits in Tabelle 1 beschrieben, durch mehrere Bearbeitungsstufen mit gegebenenfalls mehreren verfügbaren Maschinen charakterisiert. Er stellt daher eine Kombination aus einem parallelen Maschinenproblem und einer Flow Shop-Umgebung dar, was sich in der Komplexitätshierarchie (Abb. 1) widerspiegelt. Eine schematische Darstellung eines hybriden Flow Shops ist in Abbildung 3 zu sehen. Dieser besteht analog zum Beispiel aus Tabelle 1 aus zwei Bearbeitungsstufen mit vier identischen Maschinen auf der ersten und zwei unabhängigen auf der zweiten Stufe. Dazwischen befindet sich ein Puffer mit unbeschränkter Größe.

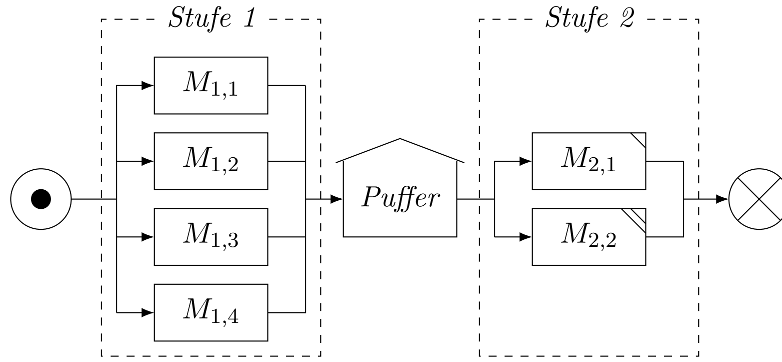


Abbildung 3: Schematische Darstellung eines beispielhaften hybriden Flow Shops.

In Mäckel et al. [3] wurde bereits gezeigt, dass Algorithmen auf in der Komplexitätshierarchie niedrigere Probleme ebenfalls anwendbar sind, sodass das entwickelte Framework automatisch auch auf Flow Shops, parallele Maschinen und Einmaschinenprobleme anwendbar ist. Dies folgt direkt aus der Reduktion der Probleme durch Weglassung von Bearbeitungsstufen oder der Reduktion der Maschinenzahl auf eins. Wie bereits erwähnt, werden zusätzlich zur hybriden Flow Shop-Umgebung die Nebenbedingungen *skip* und M_j umgesetzt. Die Zielfunktion wird variabel gehalten, in der Auswertung wird jedoch ausschließlich der *Makespan* verwendet. Die Graham-Notation des in dieser Arbeit betrachteten Problems ergibt sich damit als

$$FHm, ((RM^{(i)})_{i=1}^{m}) \mid skip, M_j \mid C_{max} \tag{1}$$

Zur Lösung des Maschinenbelegungsproblems wird ein Evolutionärer Algorithmus implementiert. Dieser generiert einen Ablaufplan für eine gegebene Probleminstanz. Die Generierung einer Startlösung, die anschließend iterativ verbessert wird, ist Teil des Algorithmus. Die Grundlagen für den Aufbau und die verwendeten Operatoren werden im folgenden Abschnitt beschrieben.

2.2 Evolutionäre Algorithmen

In den 1950er Jahren wurden zum ersten Mal Versuche unternommen, die natürliche Evolutionstheorie nach Darwin für technische Anwendungen zu adaptieren [15, S. 1][16, S. 14]. Zu dieser Zeit waren heuristische Ansätze, das heißt Algorithmen, die ein gegebenes Problem nicht garantiert optimal lösen [17, S. 55], noch nicht breit akzeptiert und standen im direkten Konflikt mit der mathematischen Optimierung, für die beispielsweise Konvergenzbeweise möglich waren [18, S. 9]. Auf Grund von schwacher Hardware konnten auch die Lösungsgeschwindigkeiten zu dieser Zeit noch nicht überzeugen [15, S. 2]. Einen Aufschwung erzielte das Forschungsfeld in den 1990er Jahren nach der Publikation des Buches *Genetic algorithms in search, optimization, and machine learning* [19][15, S. 3]. Der Begriff „Evolutionärer Algorithmus“ bezeichnet die gesamte Gruppe von Ansätzen, die sich an Analogien aus der Evolutionstheorie bedienen [20, S. 9]. Evolutionäre Algorithmen unterscheiden sich dabei in einigen Punkten fundamental von anderen Lösungsstrategien [19, S. 7]:

- EA nutzen eine Kodierung des Parameterraums.
- EA arbeiten mit einer Menge von Lösungen anstatt nur einem Punkt.
- EA benötigen kein Wissen über die Struktur der Zielfunktion wie zum Beispiel Ableitungen, sondern arbeiten allein mit Funktionsauswertungen.
- EA nutzen Operatoren, die in Teilen dem Zufall unterliegen, anstatt mathematisch-deterministischer Übergangsregeln.

Für die Operationen und Strukturen eines EA werden häufig Analogien zur Evolutionstheorie gezogen [15, S. 3 f.][16, S. 13 f.]:

- Eine Lösung wird als *Individuum* bezeichnet.
- Die Menge aller aktuell betrachteten Lösungen bildet die *Population*.
- Operatoren auf den Lösungen werden *Variationsoperatoren* genannt. Diese lassen sich in zwei Gruppen unterteilen:
  - *Mutationsoperatoren* ändern ein Individuum.
  - *Rekombinations-* oder *Crossoveroperatoren* erzeugen einen Nachkommen aus mehreren Individuen.
- *Selektionsoperatoren* wählen eine Untermenge von Individuen aus der Population aus.

EA lassen sich nach der Definition von Sörensen et al. [17, S. 52][21, S. 6] als Metaheuristik kategorisieren, da sie ein allgemeines Framework zur Entwicklung von problemspezifischen Lösungsalgorithmen bilden. Sie bieten eine sehr robuste Basis und lassen sich durch ihre allgemeine Struktur auf neue Probleme einfacher übertragen [16, S. 20 f.]. EA lassen sich insbesondere sinnvoll einsetzen, wenn keine Informationen über die Topologie der Zielfunktion oder genauere Systemeigenschaften vorliegen. Im Folgenden wird nun der Aufbau eines Evolutionären Algorithmus beschrieben und auf einige gängige Operatoren eingegangen, auf die im Implementierungsteil der Arbeit in Abschnitt 4 Bezug genommen wird.

2.2.1 Aufbau

Der Aufbau eines Evolutionären Algorithmus ist in Abbildung 4 in Anlehnung an Pétrowski et al. [15, Abb. 1.1] und Eiben et al. [16, S. 26 f.] als Flussdiagramm gezeigt.

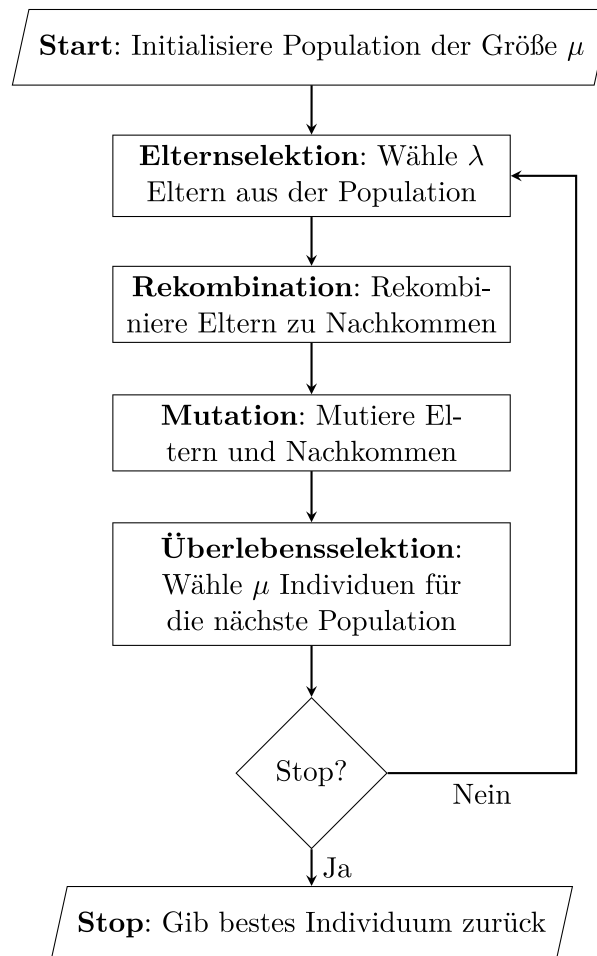


Abbildung 4: Schematischer Aufbau eines Evolutionären Algorithmus.

Bei dem Algorithmus handelt es sich nach der Definition von Sörensen et al. [17, S. 56 f.] speziell um eine populationsbasierte Metaheuristik. Das Verfahren wendet nach Generierung der Menge an Startlösungen die beiden Variationsoperatoren *Rekombination* und *Mutation* auf die Population an. Dies geschieht in einer iterativen Vorgehensweise. Mit jeder Iteration wird die aktuelle Population in eine neue überführt. Die Variationsoperatoren sorgen dabei zugleich für eine Diversifizierung der Individuen, als auch für das Auffinden besserer Lösungen (Konvergenz) [16, S. 26]. Die Überlebensselektion am Ende der Schleife sorgt für eine konstante Populationsgröße und steuert durch die Art der Wahl maßgeblich die Diversität und Konvergenz der Lösungsmenge [16, S. 26]. Die Grundelemente *Selektion*, *Rekombination* und *Mutation* werden in den folgenden Abschnitten beschrieben.

2.2.2 Eltern- und Überlebensselektion

Ein *Selektionsoperator* wird an zwei Stellen des Algorithmus eingesetzt. Zum einen müssen in jeder Iteration Individuen zur *Rekombination* und *Mutation* ausgewählt werden, zum anderen wird die Größe der Population am Ende jeder Schleife auf eine konstante Größe beschränkt. Konzeptuell wird an beiden Stellen aus einer Population eine Untermenge ausgewählt, dennoch kann nicht jeder Selektionsoperator an beiden Stellen verwendet werden. Für die Überlebensselektion ist es nötig, dass genau μ Individuen für die nächste Generation gewählt werden, um die Populationsgröße über die Iterationen konstant zu halten, während die Anzahl der gewählten Individuen für die Variation prinzipiell beliebig sein darf. Des Weiteren darf maximal einer der Operatoren eine zufällige Wahl durchführen, da sonst kein Selektionsdruck gegen das Optimum entsteht. Andererseits sollten die Operatoren gute Individuen nicht zu stark bevorzugen, um nicht gegen ein potenzielles lokales Optimum zu konvergieren [15, S. 5 f.]. Es ist weiterhin nötig, eine Lösungsdiversität in der Population zu erhalten, um das volle Potenzial der Variationsoperatoren, insbesondere der *Rekombination* (siehe Abschnitt 2.2.5), zu entfalten [16, S. 26]. Das Phänomen, dass im Laufe der Optimierung alle Individuen durch Variation identisch werden und in einem lokalen Optimum enden, nennt man *Genetic Drift* [15, S. 7 ff.]. Es ist daher nötig, je nach Ausprägung der Zielfunktion, die Selektionsoperatoren derart zu wählen und ggf. zu parametrisieren, dass ein Selektionsdruck gegen das Optimum besteht, um die Konvergenz der Population zu fördern, aber auch eine Lösungsdiversität zu erhalten, um aus lokalen Optima zu entkommen. Es werden nun drei Selektionsmethoden aus der Literatur vorgestellt, die in der Arbeit implementiert und evaluiert wurden [16]:

Zufällige Wahl Die zufällige Wahl der Individuen bietet sich für die Elternselektion an, um eine unverzerrte Untermenge der Population für die Variation zu erhalten. Diese Methode fördert die gleichmäßige Suche über den Suchraum, der Selektionsdruck ist hier jedoch unbestimmt. Der Algorithmus kann in zwei Ausprägungen implementiert werden. Variante 1 (Alg. 1) wählt eine konstante Anzahl $n_{sel} \leq \mu$ Individuen zufällig (mit oder ohne Zurücklegen) aus der Population aus. Variante 2 (Alg. 2) wählt jedes Individuum mit einer Selektionswahrscheinlichkeit $p_{sel} \in [0, 1]$ aus. Die Anzahl der zu wählenden Individuen für Variante 1 kann auch aus der Selektionswahrscheinlichkeit ermittelt werden, indem $n_{sel} = \lceil \mu * p_{sel} \rceil$ angenommen wird.

Algorithmus 1 Selektionsoperator zufällige Wahl mit fester Anzahl aus der Literatur.

Require: Population P; Anzahl zu wählender Individuen n_{sel}.

1: $\hat{P} \leftarrow \varnothing$

2: **repeat** n_{sel}

3: $\quad i \leftarrow random(0, |P|)$

4: $\quad \hat{P} \leftarrow \hat{P} \cup P[i]$

5: $\quad$ **if** $\neg$ zurücklegen **then**

6: $\quad\quad P \leftarrow P \setminus P[i]$

7: $\quad$ **end if**

8: **end**

9: **return** $\hat{P}$

Algorithmus 2 Selektionsoperator zufällige Wahl mit stochastischer Anzahl aus der Literatur.

Require: Population P; Wahrscheinlichkeit der Auswahl p_{sel}.

1: $\hat{P} \leftarrow \varnothing$

2: **for** Individuum $p \in P$ **do**

3: $\quad$ **if** $random(0, 1) \leq p_{sel}$ **then**

4: $\quad\quad \hat{P} \leftarrow \hat{P} \cup p$

5: $\quad$ **end if**

6: **end for**

7: **return** $\hat{P}$

Beide Varianten lassen sich für die Elternselektion einsetzten. Für die Überlebensselektion ist eine konstante Populationsgröße Voraussetzung, weshalb nur Variante 1 mit $n_{sel} = \mu$ eingesetzt werden kann. Wichtig ist hier, dass bei zufälliger Wahl das beste Individuum verloren gehen kann. Diese Problematik besteht für viele stochastische Wahlverfahren. Um dies zu verhindern, kann entweder die beste bisher gefundene Lösung gespeichert oder ein elitistischer Ansatz verwendet werden. Durch Speichern der besten Lösung wird

sichergestellt, dass Optima nicht unwiederbringlich verlassen werden. Der Rückschritt des Algorithmus kann bei hoch multimodalen Problemen helfen, nicht in lokalen Optima stecken zu bleiben [15, S. 19 f.]. Bei elitistischen Ansätzen wird nach der Überlebensselektion das beste Individuum der Ursprungspopulation wieder hinzugefügt, sollte es verloren gegangen sein [15, S. 19]. Genauer gesagt wird das ursprünglich beste Individuum wieder hinzugefügt, wenn das neue beste Individuum einen schlechteren Zielfunktionswert bezüglich des Optimierungsziels aufweist, da mehrere Individuen denselben Zielfunktionswert haben können und daher keine totale Ordnung der Elemente vorausgesetzt werden kann. Das zu entfernende Individuum kann zum Beispiel das schlechteste der Population sein oder zufällig gewählt werden. Das Vorgehen ist in Algorithmus 3 gezeigt.

Algorithmus 3 Elitistisches Framework für Selektionsoperatoren aus der Literatur.

Require: Population P. Zu maximierende Zielfunktion f.

1: $p \leftarrow \text{argmax}_{p \in P}\{f(p)\}$
2: $\hat{P} \leftarrow$ Führe Selektionsoperator aus.
3: $\hat{p} \leftarrow \text{argmax}_{\hat{p} \in \hat{P}}\{f(\hat{p})\}$
4: **if** $f(p) > f(\hat{p})$ **then**
5: $\quad \tilde{p} \leftarrow$ Wähle zu entfernendes Individuum aus $\hat{P}$.
6: $\quad \hat{P} \leftarrow \hat{P} \setminus \tilde{p}$
7: $\quad \hat{P} \leftarrow \hat{P} \cup p$
8: **end if**
9: **return** $\hat{P}$

Die neutrale Wahl von Individuen stärkt die Diversität der Population. Um Konvergenz des Algorithmus zu erzielen, muss jedoch mindestens einer der Selektionsschritte bessere Individuen gemäß der Zielfunktion des Optimierungsproblems bevorzugen. Die zufällige Wahl sollte daher in höchstens einem der beiden Selektionen zum Einsatz kommen.

Truncation selection Bei der *Truncation selection* werden bei der Elternselektion die n_{sel} und bei der Überlebensselektion die μ besten Individuen aus der Population gewählt [15, S. 18]. Dies sorgt für einen sehr hohen Selektionsdruck, da schlechte Individuen sehr schnell aus der Population verdrängt werden. Im Falle der Elternselektion können schlechte Individuen nur dann berücksichtigt werden, wenn durch die folgende Überlebensselektion bessere Individuen verdrängt werden und sie dadurch in der Hierarchie steigen. Populär sind bei der Überlebensselektion die zwei Varianten (μ, λ) und $(\mu + \lambda)$. Bei (μ, λ) werden die Individuen nur aus der Gruppe der bereits variierten Individuen gewählt, wofür $\lambda \geq \mu$ nötig ist. Bei $(\mu+\lambda)$ werden die Individuen aus der Ausgangspopulation mit den Variierten gemischt und darauf die besten gewählt. (μ, λ) ist dabei nicht elitistisch und das beste Individuum kann verloren gehen [15, S. 18]. Das Vorgehen der beiden Varianten ist in Algorithmus 4 gezeigt.

Algorithmus 4 $(\mu \overset{+}{,} \lambda)$ Selektion aus der Literatur.

Require: Population P; Variierte Population P'; Anzahl zu wählender Individuen n_{sel}. Zu maximierende Zielfunktion f.

1: $\hat{P} \leftarrow \varnothing$
2: $\tilde{P} \leftarrow P'$

3: **if** $'+'$-Selektion **then**
4: $\quad \tilde{P} \leftarrow \tilde{P} \cup P$
5: **end if**

6: $\tilde{P} \leftarrow$ Sortiere $\tilde{P}$ absteigend nach Zielfunktionswert f.
7: **for** $i \in 0 \ldots n_{sel} - 1$ **do**
8: $\quad \hat{P} \leftarrow \hat{P} \cup \tilde{P}[i]$
9: **end for**

10: **return** $\hat{P}$

Turnierselektion Bei der Turnierselektion werden iterativ zufällige Gruppen von Individuen gewählt und davon das Beste in die neue Population übernommen. Die Turnierselektion bevorzugt dabei klar bessere Individuen, hat aber im Gegensatz zu Fitness- oder Rang-proportionaler Selektion den Vorteil, dass es schneller berechenbar und invariant gegen additive Verschiebungen ist [15, S. 9–17]. Der Algorithmus wählt n_{sel}-Mal eine zufällige Gruppe der Größe $\kappa < \mu$ aus und fügt das beste Individuum dieser Gruppe der neuen Population hinzu. Der Algorithmus ist in Alg. 5 gezeigt.

Algorithmus 5 Selektionsoperator Turnierselektion aus der Literatur.

Require: Population P; Anzahl zu wählender Individuen n_{sel}; Zu maximierende Zielfunktion f; Gruppengröße κ.

1: $\hat{P} \leftarrow \varnothing$

2: **repeat** n_{sel}
3: $\quad Q \leftarrow \varnothing$
4: $\quad$ **repeat** κ
5: $\quad\quad i \leftarrow random(0, |P|)$
6: $\quad\quad Q \leftarrow Q \cup P[i]$
7: $\quad$ **end**
8: $\quad p \leftarrow \text{argmax}_{p \in Q}\{f(p)\}$
9: $\quad \hat{P} \leftarrow \hat{P} \cup p$
10: **end**

11: **return** $\hat{P}$

Über den Parameter κ lässt sich der Selektionsdruck steuern. Bei großem Kappa werden die gewählten Gruppen größer und die Wahrscheinlichkeit, dass schlechte Individuen ausgewählt werden, sinkt. Dennoch handelt es sich um ein nicht-elitistisches Verfahren, bei dem das beste Individuum bei der Überlebensselektion verloren gehen kann. Fang et al. [22, S. 4] untersuchten eben diesen *Not-Sampled Issue* bei dem es um das Problem der nicht-Betrachtung von Individuen aus der Population geht. Xie et al. [23, S. 3] geben entsprechend die Wahrscheinlichkeit, dass ein Individuum aus einer Population der Größe N bei μ-maliger Turnierselektion der Größe κ mit Zurücklegen mindestens einmal in einer Gruppe ausgewählt wurde als

$$P_{sel} = 1 - \left(\left(\frac{N-1}{N}\right)^N\right)^{\frac{\mu}{N} * \kappa}$$

an. Daraus folgt eine Wahrscheinlichkeit von

$$1 - P_{sel} = \left(1 - \frac{1}{N}\right)^{\kappa * \mu}$$

dass das beste Individuum nicht gewählt wurde und somit bei der Überlebensselektion verloren geht. Dies setzt sich zunächst zusammen aus der Wahrscheinlichkeit $1 - \frac{1}{N}$ bei einmaligem Ziehen nicht gewählt zu werden, potenziert mit κ Zügen. Dies wird μ-Mal wiederholt, wobei das beste Individuum natürlich bei einer Wahl garantiert überleben würde. Wegen

$$\lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n = e$$

gilt für große N und μ, $1 - P_{sel} \approx e^{-\kappa}$. Die Wahrscheinlichkeit, dass das beste Individuum verloren geht, fällt also bei steigender Gruppengröße κ exponentiell ab.

2.2.3 Repräsentation

In den folgenden Abschnitten werden ausschließlich Operatoren für die Klasse der Permutationsprobleme $\mathbb{P}_n$ adressiert, da diese Kodierung für die betrachteten Maschinenbelegungsprobleme verwendet werden kann [16, S. 68]. Des Weiteren werden ausschließlich Operatoren beschrieben, die im Implementierungsteil dieser Arbeit (Abschnitt 4) Anwendung finden. Ein Individuum wird typischerweise als String, genannt *Chromosom*, aus Werten, genannt *Gen*, repräsentiert. Dazu ist eine Abbildung inklusive einer Abstraktion des Ursprungsproblems (*Phänotyp*) auf die Repräsentation im Chromosom (*Genotyp*) nötig. Für Permutationsprobleme besteht ein Chromosom aus einer Folge einzigartiger Elemente, typischerweise in der Genotyp-Phänotyp-Abbildung als ganze Zahl repräsentiert. Eine beispielhafte Repräsentation der Elemente 1-8 ist in Abbildung 5 in Anlehnung an die Darstellung von Eiben et al. [16] gezeigt. Diese wird fortlaufend als Beispiel für die Variationsoperatoren genutzt.

2	6	4	1	5	3	8	7

Abbildung 5: Repräsentation eines Individuums für Permutationsprobleme.

Die Veränderung der Chromosomen der Individuen findet mittels *Rekombinations-* und *Mutationsoperatoren* statt. Die Auswahl der Operatoren hat dabei einen Einfluss auf die Lösungsgüte und Konvergenzgeschwindigkeit. Insbesondere sollten die Operatoren einen möglichst großen Suchraum erreichen können. Variationsoperatoren haben die Aufgaben, den Suchraum breit zu erkunden, aber auch an vielversprechenden Punkten gegen das Optimum zu konvergieren [15, S. 22]. Wenn keine Annahmen über die Suchrichtung, zum Beispiel über Gradientenapproximationen vorliegen, müssen die Operatoren des Weiteren unverzerrt arbeiten, um keine Suchrichtung zu bevorzugen. Die Erreichbarkeit hängt unter anderem von der Kodierung der Individuen ab. Pétrowski et al. [15, S. 5 f.] betonen die Wichtigkeit der Abstimmung von Problem, Lösungsrepräsentation und Auswahl der Variationsoperatoren.

2.2.4 Mutation

In dieser Arbeit werden zwei Mutationsoperatoren betrachtet, die das Chromosom eines Individuums ändern und die neue Lösung zurückgeben.

2-Swap Der *2-Swap* wählt zwei zufällige Gene a, b aus dem Chromosom π aus und tauscht ihre Positionen aus [16, S. 69]. Je nach Implementierung kann $a \neq b$ vorausgesetzt werden, für große Chromosomen kann es aber effizienter sein, diese Bedingung nicht zu überprüfen. Eine beispielhafte Operation ist in Abbildung 6 gezeigt. Der Algorithmus ist in Alg. 6 skizziert.

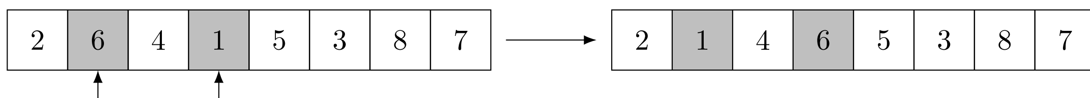


Abbildung 6: 2-Swap Mutation eines Individuums.

Algorithmus 6 Mutationsoperation 2-Swap aus der Literatur.

Require: Chromosom π.

1: $a \leftarrow random(0, |\pi|)$
2: $b \leftarrow random(0, |\pi|)$
3: $save \leftarrow \pi[a]$
4: $\pi[a] \leftarrow \pi[b]$
5: $\pi[b] \leftarrow save$
6: **return** π

Translokation/Einfügemutation Die zweite betrachtete Mutationsoperation ist die *Translokation* oder *Einfügemutation*. Dabei wird ein zufälliges Gen an eine zufällige Stelle im Chromosom verschoben [16, S. 69]. Eine beispielhafte Operation ist in Abbildung 7 gezeigt. Der Algorithmus ist in Alg. 7 skizziert.

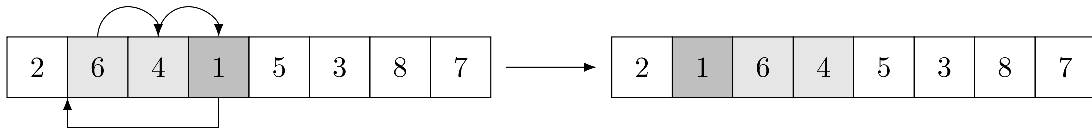


Abbildung 7: Translokations-Mutation eines Individuums.

Algorithmus 7 Mutationsoperation Translokation aus der Literatur.

Require: Chromosom π.

1: $a \leftarrow random(0, |\pi|)$
2: $b \leftarrow random(0, |\pi|)$

3: **if** $a \neq b$ **then**
4:     $save \leftarrow \pi[a]$
5:     **if** $b > a$ **then**
6:         **for** $i \in a \ldots b-1$ **do**
7:             $\pi[i] \leftarrow \pi[i+1]$
8:         **end for**
9:     **else**
10:         **for** $i \in a \ldots b+1$ **do** ▷ Absteigende Iteration, da $a > b$
11:             $\pi[i] \leftarrow \pi[i-1]$
12:         **end for**
13:     **end if**
14:     $\pi[b] \leftarrow save$
15: **end if**

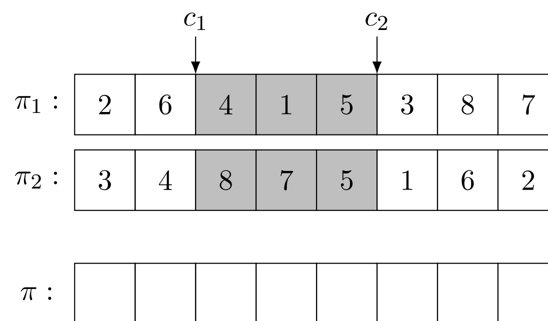
16: **return** π

2.2.5 Rekombination

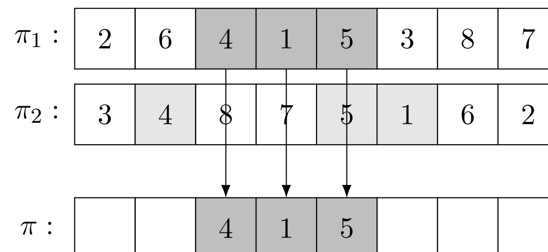
Rekombinationsoperatoren erzeugen ein neues Individuum aus mehreren, meist zwei Eltern. Die Operation der Verschmelzung wird häufig als *Crossover* bezeichnet [16, S. 32]. Crossoveroperatoren erfüllen dabei meist die beiden folgenden Eigenschaften [15, S. 22 f.]:

- Das Crossover zweier identischer Chromosome resultiert im selben Chromosom.
- Zwei Individuen, die im Suchraum bei definiertem Abstandsmaß nah beieinander liegen, generieren ein neues Individuum, das nah bei den Eltern liegt.

1. Wähle Crossoverpunkte $c_1, c_2 \in [0, |\pi|]$ mit $c_1 < c_2$.



2. Kopiere Gene aus π_1 zwischen c_1 und c_2 nach π.



3. Kopiere restliche Gene aus π_2 von links nach rechts beginnend bei c_2.

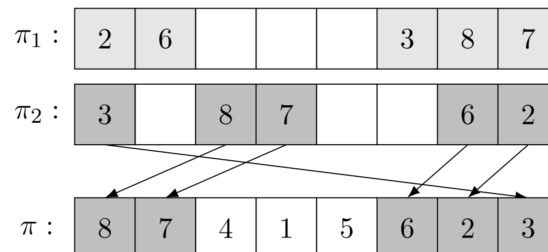


Abbildung 8: Order-Based-Crossover.

Im Permutationsraum ist ein Crossover schwieriger als für Probleme im $\mathbb{R}^n$, da jedes Element nur einmal vorkommen darf und dadurch der simple Austausch von Teilen

des Chromosoms zu unzulässigen Individuen führt [16, S. 70]. In dieser Arbeit wird eine Variante des Order-Based-Crossover betrachtet, dass von Davis [24] speziell für Permutationsprobleme entwickelt wurde. Das Vorgehen wird anhand des Beispiels in Abb. 8 erläutert. Es ist zu sehen, dass der Algorithmus, die Permutation und die Position der Gene zwischen den Crossoverpunkten c_1 und c_2 für Chromosom π_1 beibehält. Die Gene in π_2 werden in ihrer Position verändert, behalten aber ihre relative Reihenfolge bei. Der Algorithmus ist in Alg. 8 gezeigt.

Algorithmus 8 Order-Based-Crossover aus der Literatur.

Require: Elternchromosomen π_1, π_2.

```
1:  π ← array(|π1|)
2:  ▷ Erzeuge zufällige Crossoverpunkte
3:  r1 ← random(0, |π|)
4:  r2 ← random(0, |π|)
5:  ▷ Stelle c1 ≤ c2 sicher
6:  c1 ← min(r1, r2)
7:  c2 ← max(r1, r2)
8:  ▷ Kopiere Gene zwischen c1 und c2 aus π1
9:  for i ∈ [c1, c2 − 1] do
10:     π[i] ← π1[i]
11: end for
12: ▷ Kopiere Gene aus π2 beginnend bei c2
13: k ← c2
14: for i ∈ [0, |π| − 1] do
15:     j ← (i + c2) mod |π|
16:     if π2[j] ∉ π then
17:         π[k] ← π2[j]
18:         k ← (k + 1) mod |π|
19:     end if
20: end for
21: return π
```

2.3 Parameteroptimierung

Ziel der Parameteroptimierung (auch *Parameter-Tuning* oder *Algorithmenkonfiguration* genannt [25, S. 1]) ist das Finden einer guten Parametrisierung eines parameterbehafteten Algorithmus [26, S. 1]. Die Wahl guter Parameter beeinflusst dabei häufig signifikant die Performanz der Algorithmen und kann problemspezifisch variieren. Die automatisier-

te Konfiguration von Lösungsverfahren stellt daher einen wichtigen Teil der Suche nach geeigneten Algorithmen dar [25, S. 1]. Da die optimalen Parameter und auch ihr Einfluss und ihre gegenseitige Abhängigkeit nicht bekannt sind, handelt es sich um ein Black-Box-Problem. Die Zielfunktion liegt also nicht in einer geschlossenen Form vor und kann daher nicht analytisch gelöst werden. Die optimalen Parameter können nur durch Einsetzen von Versuchspunkten gefunden werden, wobei das Verhalten der Zielfunktion von modellbasierter Optimierung iterativ approximiert werden kann. Man unterscheidet zwischen iterativen und nicht-iterativen Parameteroptimierungsverfahren [6]. Nicht-iterative Optimierung legt die zu testenden Parametervektoren zu Beginn der Suche fest, während die iterative Suche neue Punkte auf Grund bereits untersuchter Parametrisierungen auswählt.

2.3.1 Gittersuche

Bei der Gittersuche handelt es sich um das klassische Verfahren des *Design of Experiments (DoE)* [27], das häufig auch als „voll faktorielles Design“ bezeichnet wird [28, S. 1 f.][29, S. 83 ff.] und zur systematischen Suche in einem gegebenen Parameterraum angewendet werden kann [26, S. 1]. Der Wertevektor jedes Parameters wird dabei vor der Optimierung festgelegt, sodass es sich bei dem Verfahren um eine nicht-iterative Suche handelt [6, S. 25]. Für kategorische Parameter entspricht dieser der Menge der möglichen Werte. Für numerische Parameter können die Werte im zulässigen Wertebereich gewählt werden. Die Wahl kann beispielsweise äquidistant, das heißt linear, oder auch logarithmisch skaliert erfolgen [28, S. 3]. Die zu testenden Wertepaare ψ ergeben sich aus der Potenzmenge der Parametervektoren $\psi_1 \dots \psi_1$.

$$|\psi| = |\psi_1| \times \ \dots \ \times |\psi_n| = \prod_{k=1}^{n} |\psi_k|$$

Werden die Parameter äquidistant gewählt, können in der Auswertung der Ergebnisse die Einflüsse der einzelnen Parameter genau analysiert werden. Dies kann beispielsweise dazu dienen, für weitere Optimierungen irrelevante Parameter nach dem Konzept des Parameter-Screenings auszuschließen, oder den Wertebereich drastisch einzuschränken, um den Suchraum zu verkleinern und so eine feinere Nachoptimierung zu ermöglichen [6, S. 9]. Die äquidistante Wahl bietet sich vor allem dann an, wenn kein Wissen über den Einfluss des Parameters auf die Performanz des zu konfigurierenden Algorithmus vorliegt. Der Vergleich aller möglichen Kombinationen führt zu einer hohen Anzahl an Experimenten [27, S. 14], dennoch ist die Gittersuche durch ihre Einfachheit und insbesondere Unvoreingenommenheit bezüglich der zu testenden Werte beliebt [26, S. 1]. Die Gittersuche wird in dieser Arbeit durch ihre simple Algorithmik selbst implementiert und die Ergebnisse in Abschnitt 5.3 besprochen.

2.3.2 Modellbasierte Parameteroptimierung

Iterative Parameteroptimierungsmethoden verringern die Anzahl der zu testenden Versuchspunkte, indem sie während der Suche iterativ den nächsten Punkt auf Grund der bereits gewonnenen Erkenntnisse ermitteln. Prominentester Vertreter unter den nicht-modellbasierten Verfahren ist der auf *racing* basierende F-RACE Algorithmus [30][31, S. 2]. Moderne modellbasierte Parameteroptimierung nutzt *Kriging* bzw. *Gaußprozess-Regression* [25], um auf Grund von gesampelten Punkten ein Modell der Zielfunktion zu approximieren. Beide schätzen dabei nicht nur den Verlauf der Zielfunktion, sondern auch den Fehler der Approximation [32, S. 139]. Dies ermöglicht es, ein Gleichgewicht zwischen der Exploration des Suchraumes auf Grund der Fehlerschätzung und der Konvergenz zum Optimum auf Grund der Zielfunktionsschätzung zu finden. Die Approximation eines Modells verringert weiter die Anzahl der zu testenden Punkte, allerdings auf Kosten der Berechnung des Metamodells von $\mathcal{O}(N)$ pro Iteration, wobei N die Anzahl der gesampelten Punkte darstellt [32, S. 139]. Weiterhin kann bei bayes'scher Optimierung [33] statistische Varianz aus den Zufallskomponenten der Algorithmen als Rauschen der Zielfunktion modelliert werden [34, S. 1]. Die Auswahl neuer Punkte erfolgt mittels einer wählbaren *Aquisitionsfunktion*. In dieser Arbeit werden die folgenden drei Funktionen verwendet, die gleichzeitig die populärsten Aquisitionsfunktionen darstellen [35, S. 3]:

- **Probability of Improvement (PI)**: Wähle den Punkt, an dem die Wahrscheinlichkeit für eine Verbesserung des aktuellen Optimums am höchsten ist. PI bezieht dabei nicht die Größe der zu erwartenden Verbesserung mit ein, sodass sehr wahrscheinlich eine Verbesserung erzielt wird, die Konvergenzgeschwindigkeit, sowie das Entkommen aus lokalen Minima aber nicht gesteuert wird.

- **Expected Improvement (EI)**: Wähle den Punkt, an dem die erwartete Verbesserung des aktuellen Optimums am höchsten ist.

- **Lower Confidence Bound (LCB)** Wähle den Punkt, der unter optimistischer Annahme den besten Zielfunktionswert liefert. LCB enthält einen zusätzlichen Parameter κ, der zwischen Exploration und Konvergenz gewichtet.

Da die Auswahl der Aquisitionsfunktionen erneut ein Optimierungsproblem darstellt und nicht klar ist, welche für den gegebenen Anwendungsfall die Beste ist [35, S. 3–6], kann auch eine der obigen probabilistisch gewählt werden, wobei die Wahrscheinlichkeitsverteilung während der Optimierung angepasst werden kann. Für die modellbasierte Parameteroptimierung wird in dieser Arbeit das Framework *scikit-optimize (skopt)* [36] genutzt, das zu den am weitesten verbreiteten Frameworks zählt und auf *scikit-learn (sklearn)* [37], *scipy* [38] und *numpy* [39] aufbaut. Dieses optimiert parallel auch die Wahrscheinlichkeitsverteilung für die Wahl der Aquisitionsfunktion, sowie den Parameter κ. Die Anwendung des Frameworks wird in Abschnitt 5.4 besprochen.

2.4 Kombinatorische Logik und CLS

Zur automatisierten Synthese der Algorithmenkompositionen für die Parameteroptimierung wird das am Lehrstuhl 14 der Informatik der TU Dortmund entwickelte Syntheseframework *Combinatory Logic Synthesizer* (CLS) in seiner Python-Implementierung verwendet. CLS kann eingesetzt werden, um Software aus Einzelkomponenten, sogenannten *Kombinatoren*, zusammenzusetzen [5]. Es basiert auf endlicher kombinatorischer Logik (FCL) mit Intersektionstypen [40]. Im Vergleich zu klassischer kombinatorischer Logik benötigt es lediglich ein Axiom (5) und *modus ponens* (4) [40, S. 8 f.] als Ableitungsregel. Diese wird dadurch entscheidbar und EXPTIME-vollständig [40, S. 5], was eine automatisierte Inhabitation (8) ermöglicht. Kombinatorische Logik ist stark verwandt mit dem getypten Lambda-Kalkül [41, S. 87 ff.], dessen Notation zur präziseren Beschreibung der Typen verwendet wird.

Vorarbeiten zur Synthese von konstruktiven Maschinenbelegungsalgorithmen wurden in der vorangegangenen Bachelorarbeit gezeigt [2, 3]. Eine Erweiterung um weitere Algorithmen, sowie dem dynamischen Aufbau des Komponentenrepositorys wurde in der Projektgruppe Autoschedule [4] entwickelt. Zur Synthese von Maschinenbelegungsalgorithmen werden diese zunächst in Komponenten zerlegt, die eine gemeinsame Schnittstelle definieren, um sie austauschbar zu machen. Eine Algorithmenkomposition ist die Applikation verschiedener Komponenten aus der Komponentensammlung Γ. Die Definition der Schnittstellen findet in Form eines Typen τ der Komponente x statt. Ein Typ τ kann dabei entweder eine Konstante a, eine Variable α, oder ein Funktionstyp (Pfeiltyp) sein (2)[5, S. 28]. Terme bestehen entweder aus getypten Kombinatoren $(X : \rho) \in \Gamma$ oder einer Applikation [5, S. 28] (3)

$$\text{Typen } \tau, \tau' ::= a \mid \alpha \mid \tau \to \tau' \tag{2}$$

$$\text{Terme } e, e' ::= X \mid (e, e') \tag{3}$$

Komponenten mit denselben Typen können definitionsgemäß gegeneinander ausgetauscht werden, ohne die funktionale Korrektheit der applikativen Komposition zu beeinträchtigen. Eine Applikation $\tau' \to \tau$ kann dabei so verstanden werden, dass für die Benutzung des Kombinators vom Schnittstellentyp τ eine weiterer Term des Typs τ' benötigt wird, welcher wieder eine Applikation sein kann. Eine Komponente mit mehreren Voraussetzungen $\tau_1 \ldots \tau_n$ kann also mit dem linksassoziativen Typen $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$ getypt werden. Die Ableitung erfolgt mittels *modus ponens* (4)[5, S. 28]:

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash \ (e\ e') : \tau} \quad (\to \text{E}) \tag{4}$$

Komponenten ohne Voraussetzungen haben primitive Typen, die mittels des Axioms (5) abgeleitet werden können, falls eine passende Komponente im Repository vorhanden ist [5, S. 28]:

$$\frac{}{\Gamma, x : \tau \ \vdash \ x : \tau} \quad (var) \tag{5}$$

Eine Erweiterung der Grammatik stellen Intersektionstypen dar. Diese erlauben es Komponenten mehrere Typen zu haben. Als Konsequenz kann eine Komponente an Stellen mit unterschiedlichen Schnittstellentypen genutzt werden. Die Ableitungsregel (6) lautet [5, S. 28]:

$$\frac{\Gamma \ \vdash \ x : \tau_1 \qquad \Gamma \ \vdash \ x : \tau_2}{\Gamma \ \vdash \ x : \tau_1 \cap \tau_2} \quad (\cap \mathrm{I}) \tag{6}$$

Zusätzlich kann eine Subtyp-Regel (7) hinzugefügt werden, um Vererbungsstrukturen zu realisieren [40, S. 19 ff.]. Die Grammatik erweitert sich dadurch um die Regel [40, S. 22]:

$$\frac{\Gamma \vdash \ x : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash \ x : \tau} \quad (\leq) \tag{7}$$

Die gesamte Grammatik mit Introduktions- und Eliminierungsregeln ist nochmal in Abbildung 9 gezeigt.

$$\frac{}{\Gamma, x : \tau \ \vdash \ x : \tau} \quad (var) \qquad\qquad \frac{\Gamma \vdash \ x : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash \ x : \tau} \quad (\leq)$$

$$\frac{\Gamma \vdash \ (e \ e') : \tau}{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'} \quad (\to \mathrm{I}) \qquad\qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash \ (e \ e') : \tau} \quad (\to \mathrm{E})$$

$$\frac{\Gamma \ \vdash \ x : \tau_1 \qquad \Gamma \ \vdash \ x : \tau_2}{\Gamma \ \vdash \ x : \tau_1 \cap \tau_2} \quad (\cap \mathrm{I}) \qquad\qquad \frac{\Gamma \ \vdash \ x : \tau_1 \cap \tau_2}{\Gamma \ \vdash \ x : \tau_1 \qquad \Gamma \ \vdash \ x : \tau_2} \quad (\cap \mathrm{E})$$

Abbildung 9: FCL($\cap, \leq$).

Die Synthese erfolgt durch die Inhabitation der Grammatik und der Beantwortung der Frage: „Gegeben eine Komponentensammlung Γ. Existiert ein Term e mit dem geforderten Zieltypen τ, der lediglich aus Applikationen der Komponenten in Γ besteht?“ [40, S. 9]:

$$\Gamma \vdash \ e : \tau? \tag{8}$$

CLS beantwortet nicht nur die Inhabitationsfrage nach der Existenz einer solchen Applikation, sondern generiert alle Lösungen (Inhabitanten) der Grammatik, falls diese kreisfrei, also endlich, ist. Die Menge der Inhabitanten stellt alle möglichen Kombinationen der Algorithmenkomponenten unter Berücksichtigung der Schnittstellentypen dar.

3 Stand der Forschung

Im folgenden Abschnitt wird eine Übersicht über den Stand aktueller Forschung in den relevanten Themengebieten gegeben. Dabei wird zunächst auf Forschung im Bereich der Evolutionären Algorithmen für Maschinenbelegungsprobleme eingegangen. Anschließend wird die Relevanz von Parameteroptimierung hervorgehoben und typische Parameteroptimierungsverfahren, die in aktueller Forschung verwendet werden, dargelegt. Es wird insbesondere gezeigt, dass die gewählten Parameter sich für verschiedene Probleme zum Teil stark unterscheiden, sodass eine Parameteroptimierung für jede neue Problemklasse essenziell ist. Anschließend wird auf aktuelle Einsatzgebiete des Combinatory Logic Synthesisers eingegangen, der im Zusammenspiel mit dem Luigi-Framework zur Orchestrierung der Ausführung, für die Konfiguration des Evolutionären Algorithmus und eingesetzt wird. Der Abschnitt schießt mit einer Auflistung von in der Literatur verfügbaren Testdatensätzen für hybride Flow Shops.

3.1 Evolutionäre Algorithmen

Die Literaturübersicht von Gao et al. [42] zeigt, dass Evolutionäre Algorithmen in der aktuellen Forschung häufig zur Lösung von hybriden Job Shop-Problemen und deren strukturell kleineren Probleme, wie die in dieser Arbeit betrachteten hybriden Flow Shops, angewendet werden. Ruiz et al. [43] zeigen in ihrer Literaturübersicht, dass das wissenschaftliche Interesse, gemessen an der Zahl der Veröffentlichungen, in diesem Themengebiet steigend ist. Schumacher [44], Gholami et al. [45], Wilson et al. [46], Yaurima et al. [47], Ruiz et al. [48], Zandieh et al. [49], Defersha et al. [50] und Zabihzadeh et al. [51] setzten die in dieser Arbeit betrachteten Evolutionären Algorithmen für hybride Flow Shop-Probleme ein und zeigen deren Überlegenheit gegenüber anderen Heuristiken. In kürzlich erschienenen Veröffentlichungen vergleichen Nguyen et al. [52] die Performance eines Evolutionären Algorithmus und einer Partikel-Schwarm-Optimierung mit klassischen Heuristiken zur Sequenzierung von Tasks von IoT-Geräten in Big Data Umgebungen. Der vorgeschlagene Evolutionäre Algorithmus dominierte dabei die anderen Verfahren in allen Evaluationsszenarien. Saremi et al. [53] setzen einen Evolutionären Algorithmus zur Planung von Crowdbasiertem Software Development im kompetitiven Umfeld ein und erreichen eine Reduktion der Projektlaufzeiten zwischen 33% und 78%. Dazu wird das mehrkriterielle Optimierungsproblem mit einer Variante des NSGA-II gelöst, der auf Grundlage von *Non-Dominated-Sorting* Pareto-optimale Lösungen sucht [54, S. 2 f.]. Ein weiteres mehrkriterielles Flow Shop-Problem wird in Zhao et al. [55] behandelt. Auch hier konnte der Evolutionäre Algorithmus bessere Lösungen für das Problem generieren als die klassischen Ansätze. Im Bereich der Echtzeitplanung erreicht Majidi [56] in seiner Dissertation eine Verbesserung der generierten Ablaufpläne von durchschnittlich 15% im Vergleich zu zusätzlich fehleranfälliger manueller Planung unter Einbezug von Fehlertoleranz. Wegener et al. [57] wenden einen Evolutionären Algorithmus auf ein Echtzeitplanungsproblem mit

dem Ziel der Minimierung von Verspätungen an. Kakkar et al. [58] setzen erfolgreich einen Evolutionären Algorithmus zur Generierung von Zeitplänen für Kurse im Universitätskontext ein. Die experimentelle Evaluation zeigt, dass der Algorithmus schon nach wenigen Generationen einen optimalen Zeitplan unter Einhaltung der diversen Nebenbedingungen generieren konnte. Eine kürzlich erschienene Arbeit von Fan et al. [59] untersucht auch den Einfluss der Lösungsrepräsentation auf die Erreichbarkeit im Lösungsraum für einen hybriden Flow Shop. Des Weiteren wird ein Evolutionärer Algorithmus um eine an Tabu-Suche angelehnte Komponente erweitert, um eine lokale Optimierung von Individuen zu ermöglichen. Dies verbindet die Vorteile von globaler und lokaler Suche in einem Algorithmus [59, S. 2].

3.2 Parameteroptimierung

Zandieh et al. [49] zeigen, dass die Performanz von parametrisierten Algorithmen stark von den gewählten Parametern abhängt. Sie heben hervor, dass bei der Optimierung der Parameter auch die Effekte zwischen den Operatoren betrachtet werden muss, was in vielen Studien keine Beachtung findet [49, S. 2]. Um ein Parameterset für ihren Evolutionären Algorithmus für einen hybriden Flow Shop zu finden, wird *Response-Surface-Methology* eingesetzt [49, S. 2], was ein Regressionsmodell für modellbasierte Parameteroptimierung darstellt [31, S. 3]. Die Parameter werden dabei auf einer kontinuierlichen Skala gesucht. Für die Mutationswahrscheinlichkeit wird ein Wert zwischen 4% und 25%, für die Crossoverwahrscheinlichkeit ein Wert zwischen 40% und 70% und für die Populationsgröße ein Wert zwischen 46 und 200 Individuen ermittelt, abhängig von der Problemgröße. Yaurima et al. [47], Ruiz et al. [48], Jabbarizadeh et al. [60] und Engin et al. [61] hingegen setzen das klassische *full factorial design* ein, das auch in dieser Arbeit angewendet wird. Jungwattanakit et al. [62] vergleichen verschiedene Konfigurationen eines Evolutionären Algorithmus für ein hybrides Flow Shop-Problem mit zehn Bearbeitungsstufen und zwei Zielfunktionen. Eine Variante des in dieser Arbeit verwendeten Order-Based-Crossover stellt sich dabei als überlegen gegenüber den anderen Verfahren heraus. Im Bereich der Mutationen werden die Operatoren *Shift* und *Pairwise Interchange*, das äquivalent zum beschriebenen *2-Swap* ist, untersucht. *Pairwise Interchange* lieferte leicht bessere Ergebnisse als *Shift*. Als beste Populationsgröße wurde $n = 30$ ermittelt. Als weitere Parameteroptimierungsverfahren werden häufig Racing-Algorithmen wie F-RACE [30], klassische sequentielle Parameteroptimierung [63, 64] wie beispielsweise im SPOT-Framework [65] oder auch Regressionsbäume [66] genutzt. In dieser Arbeit wird durch die lange Berechnungsdauer jeder Konfigurationsauswertung modellbasierte Parameteroptimierung eingesetzt, um die Anzahl an Experimenten zu reduzieren. Hutter et al. [25] vergleichen zwei verschiedene Verfahren der modellbasierten sequentiellen Parameteroptimierung. Dabei stellt sich ein Gaußprozessmodell als robuster als Kriging heraus. Auch in dieser Arbeit wird als Regressor ein Gaußprozess genutzt.

3.3 CLS

Für die automatische Konfiguration der Algorithmen wird der Combinatory Logic Synthesizer [5] verwendet. CLS wird in der aktuellen Forschung genutzt, um Produktlinien aus denselben modularen Bausteinen zu synthetisieren. Lenz et al. [67], Winkels [68] und Graefenstein et al. [69] setzen CLS in Verbindung mit SMT-Solving ein, um Pläne zur Fabrikplanung zu synthetisieren. SMT wird eingesetzt, um die eingeschränkte Ausdrückbarkeit von Constraints in der Grammatik durch Nachfilterung der Ergebnisse zu kompensieren. Eine SMT-unterstützende Variante CLS-SMT wurde des Weiteren in Kallat et al. [70] präsentiert. CLS wird weiterhin in Mages et al. [71], Kallat et al. [72, 73] und Kallat [74] zur Synthese von Simulationsmodellen von Fabrikprozessen eingesetzt. Schäfer et al. [75] und Schäfer [76] setzen CLS schließlich zur Synthese im Bereich des Motion-Planning ein, um Bewegungsmuster von Roboterarmen zu generieren.

In dieser Arbeit wird CLS in Kombination mit dem durch die Spotify AB entwickelten Orchestrierungsframework *Luigi* [77] verwendet. Die Kombination ermöglicht es sowohl kategorische als auch numerische Parameter in die Synthese der Algorithmen in dieser Arbeit zu integrieren. Luigi ermöglicht des Weiteren die Ausführung von Pipelines, die durch CLS generiert wurden, sodass ganze Ausführungsbäume realisiert werden können [78]. In der vorangegangenen Bachelorarbeit [2, 3] und der Projektgruppe Autoschedule [4] wurde CLS bereits zur Synthese von Maschinenbelegungsalgorithmen eingesetzt. Im Unterschied zu dieser Arbeit wurden dabei alle Kombinationen generiert und ausgewertet, während hier die Wahl der Kombinatoren durch die Parameteroptimierung bestimmt wird. Des Weiteren wird der bestehende Ansatz um die numerischen Parameter ergänzt, die durch Luigi realisiert werden. Die Bestimmung der Variationspunkte wird ebenfalls automatisiert durch den Einsatz von CLS umgesetzt, sodass die Parameter für die Optimierung ohne manuelle Konfiguration bestimmt werden können.

3.4 Benchmarks

Zur späteren Evaluation der implementierten Algorithmenkomponenten werden standardisierte Benchmarks-Datensätze eingesetzt. In der Literatur verfügbare Benchmarks für verschiedene Maschinenbelegungsprobleme wurden in der Bachelorarbeit von Kordus [79] herausgearbeitet. Eine reduzierte Übersicht mit Benchmarks für die in dieser Arbeit betrachteten hybriden Flow Shops mit konkretisierter Graham-Notation sind in Tabelle 3 zu sehen.

Jahr	Author(en)		α		β	γ
1988	Wittrock	[11]	$FH3,$	$(P2^1, P3^2, P3^3)$	$skip, lag$	C_{max}
2001	Néron et al.	[80]	$FHm,$	$((PM^{(i)})_{i=1}^m)$	$-$	C_{max}
2008	Ruiz et al.	[13]	$FHm,$	$((RM^{(i)})_{i=1}^m)$	$skip, M_j, r_m, prec, lag, S_{iljk}, A_{iljk}$	C_{max}
2010	Urlings	[14]	$FHm,$	$((RM^{(i)})_{i=1}^m)$	$M_j, r_m, prec, lag, S_{iljk}, A_{iljk}$	C_{max}
2010	Naderi et al.	[81]	$FHm,$	$((PM^{(i)})_{i=1}^m)$	$skip, S_{iljk}$	C_{max}
2017	Pan et al.	[82]	$FHm,$	$((PM^{(i)})_{i=1}^m)$	ddw	$\sum(w_j' E_j^{dw} + w_j T_j^{dw})$
2020	Lang et al.	[83]	$FH2,$	$(P4^1, P5^2)$	$S_{iljk}, fmls$	$\sum T_j, C_{max}$
2022	Missaoui et al.	[84]	$FHm,$	$((PM^{(i)})_{i=1}^m)$	S_{iljk}, ddw	$\sum(w_j' E_j^{dw} + w_j T_j^{dw})$
2022	Laroque et al.	[85]	$FHm,$	$((RM^{(i)})_{i=1}^m)$	$skip, M_j, batch, prec, fmls$	C_{max}

Tabelle 3: Benchmarks für hybride Flow Shops aus der Literatur nach Kordus [79].

4 Implementierung des Syntheseframeworks

Im folgenden Abschnitt wird nun die Implementierung des Evolutionären Algorithmus und des Syntheseframeworks beschrieben. Dazu wird in 4.1 zunächst auf die Repräsentation einer Lösung eingegangen. Anschließend wird in 4.2 die Implementierung der in Abschnitt 2.2 gezeigten Komponenten des Evolutionären Algorithmus einzeln beschrieben. Zusammengenommen bilden diese einen lauffähigen Algorithmus zur Lösung eines hybriden Flow Shops. Auf die Umsetzung der automatischen Modul- und Parametersuche durch CLS, sowie der Ausführung von Parameterkonfigurationen wird in Abschnitt 4.3 eingegangen. Hier wird die Synthese des Algorithmus aus den Einzelkomponenten und auch die Parametrisierung durch CLS und Luigi genauer beleuchtet. Der Abschnitt schließt mit der Definition eines ILP zur Bestimmung der optimalen Lösungen für den verwendeten Benchmark-Datensatz.

4.1 Repräsentation der Lösung

Zur Repräsentation der Lösung wird ein Wrapper-Objekt verwendet, das sowohl den berechneten Ablaufplan, als auch alle zur Planung nötigen Daten wie Maschinenanordnung, Prozesszeiten und Maschinenqualifikationen enthält. Es wird über den gesamten Planungsprozess vom leeren Ablaufplan über die Berechnung von Zwischenlösungen, bis zum abschließenden Optimierungsergebnis einheitlich verwendet. Es stellt zur Einplanung und Änderung von Maschinenzuweisungen Schnittstellen bereit, die von den Algorithmenkomponenten zur Manipulation benutzt werden können. Eine Darstellung der Felder des Objekts in UML [86] ist in Abbildung 10 zu sehen. Eine Darstellung des vollständigen Objekts befindet sich im Anhang (Abb. 26 und Abb. 27). Alle Felder und Methoden werden entsprechen der PEP8-Konvention in Python in *snake_case* bezeichnet [87]. Indizierungen werden ebenfalls als Suffix mit Unterstrich angehängt. Für die Erzeugung sind die Maschinenkonfiguration (*machine_arrangement*) und die Prozesszeiten (p_{ilj}) nötig. Zusätzlich können Matrizen für das Überspringen von Stufen ($skip_{ij}$) und Maschinenqualifikationen (e_{ilj}) übergeben werden. Die Maschinenkonfiguration wird als Vektor der Anzahl der verfügbaren Maschinen pro Stufe angegeben:

$$[11, 5, 6]$$

würde beispielsweise elf Maschinen auf der ersten, fünf auf der zweiten und sechs auf der dritten Bearbeitungsstufe beschreiben. Eine Unterscheidung zwischen parallelen identischen Maschinen (PM) und Maschinen verschiedener (QM), ggf. auftragsbezogener (RM), Geschwindigkeiten erfolgt nicht. Es wird vom allgemeinsten Fall, den unabhängigen Maschinen (RM) ausgegangen.

Schedule
+ machine_arrangement : np.ndarray + num_stages : int + num_machines : int + num_jobs : int + p_ilj : np.ndarray + skip_ij : np.ndarray = None + e_ilj : np.ndarray = None − ___arr : np.ndarray
+ ___init___(self, machine_arrangement : np.ndarray, p_ilj : np.ndarray, skip_ij : np.ndarray = None, e_ilj : np.ndarray = None) + allocate_tight(self, stage : int, machine : int, job : int) + copy(self)

Abbildung 10: Darstellung eines Ausschnitts des Schedule-Objekts in UML.

Das Feld *num_stages* folgt aus der Länge des Vektors, *num_machines* beschreibt die maximale Anzahl der Maschinen über alle Bearbeitungsstufen. Maschinen werden für jede Stufe separat aufsteigend beginnend bei null nummeriert, sodass die Bezeichnung einer einzelnen Maschine nur in Kombination mit der Nummer der Bearbeitungsstufe möglich ist. Die Aufträge werden ebenfalls von null beginnend nummeriert. Ihre Prozesszeiten auf den einzelnen Maschinen werden in der Prozesszeitenmatrix p_{ilj} übergeben, deren Indizierung über die Bearbeitungsstufe i, die Maschine auf der Bearbeitungsstufe l und den Auftrag j erfolgt. Das Feld *num_jobs* folgt aus der Länge der dritten Dimension der Prozesszeitenmatrix. Optional können Matrizen für das Überspringen von Stufen ($skip_{ij}$) und Maschinenqualifikationen (e_{ilj}) übergeben werden. Beide sind dabei binärwertig, wobei $skip_{ij} = False$ bedeutet, dass der Auftrag j auf der Stufe i bearbeitet werden muss, und $e_{ilj} = True$, dass er auf der Maschine l bearbeitet werden kann. Falls Matrizen übergeben werden, wird überprüft, dass jeder Auftrag auf mindestens einer Stufe gefertigt wird (9) und auf jeder Stufe, die der Auftrag besucht, mindestens eine Maschine qualifiziert ist (10), um die Lösbarkeit des Optimierungsproblems zu garantieren. Werden keine Matrizen übergeben, wird das jeweilige neutrale Element verwendet. Für $skip_{ij}$ ist das die Nullmatrix 0_{ij}, für e_{ilj} die Einsmatrix $\mathbb{1}_{ilj}$.

$$\forall j \in N : \exists i \in M, skip_{ij} = False \tag{9}$$

$$\forall j \in N, \forall i \in M \land skip_{ij} = False : \exists l \in M_i, e_{ilj} = True \tag{10}$$

Das Schedule-Objekt stellt Methoden zur Einplanung und Änderung von Maschinenzuweisungen bereit, von denen in Abbildung 10 nur *allocate_tight* gezeigt ist, das die Einplanung eines Auftrages j auf Maschine l der Bearbeitungsstufe i zum frühestmögli-

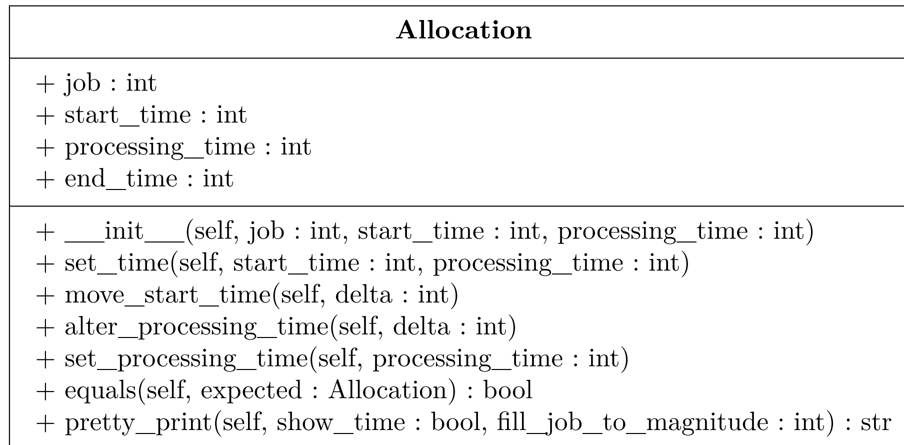


Abbildung 11: Darstellung des Allocation-Objekts in UML.

chen Startzeitpunkt vornimmt. Die Prozesszeit wird aus p_{ilj} ermittelt. Alle Maschinenzuweisungen werden im privaten Feld *__arr* mittels Zuweisungsobjekten *Allocation* (siehe Abbildung 11) gespeichert. Die Matrix hält dabei für jede Maschine, indiziert durch Bearbeitungsstufe und Maschinennummer, eine Liste an Zuweisungen. Die Zuweisungsobjekte enthalten Informationen zu Start- und Endzeitpunkten sowie Prozesszeiten, und stellen wiederum Methoden zur nachträglichen Veränderung durch die Heuristiken zur Verfügung. Die Listen sind im Normalzustand aufsteigend nach Startzeitpunkten der Zuweisungen sortiert. Die Sortierung kann während der Suche verletzt werden und, je nach Implementierung, zum Beispiel im Anschluss an einen Suchschritt wiederhergestellt werden. Bei der Erstellung und Veränderung der Zuweisung wird lediglich auf Start- und Prozesszeiten < 0 geprüft. Eine Überprüfung auf Überschneidungen der Aufträge wird erst nach Beendigung der Suche durchgeführt. Bei korrekter Implementierung der Algorithmen und ihrer Bestandteile sind diese Fälle nicht möglich, sodass die Überprüfungen lediglich als redundante Bestätigung der korrekten Funktionalität der Algorithmen dient. Eine beispielhafte Zuweisung von fünf Aufträgen auf zwei Maschinen einer Stufe ist in Abbildung 12 gezeigt.

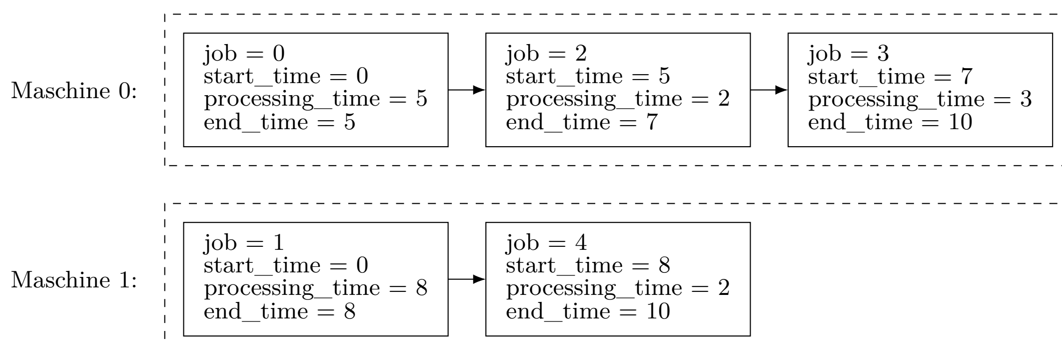


Abbildung 12: Beispiel für die Darstellung der Maschinenzuweisungen.

Die Methode *copy* des Schedule-Objekts erstellt eine tiefe Kopie inklusive aller Maschinenzuweisungen, während *shallow_copy* nur die Datenmatrizen kopiert. Das neue Objekt besitzt entsprechend keine Maschinenzuweisungen. Ein Schedule-Objekt dient als Eingabe und Zwischenformat aller Algorithmen und Komponenten. Dies erleichtert die Schnittstellenbildung während der Komponentisierung, die für den betrachteten Evolutionären Algorithmus im Folgenden beschrieben wird.

4.2 Komponentisierung des Evolutionären Algorithmus

Ein Evolutionärer Algorithmus besteht aus den in Abbildung 4 in Abschnitt 2.2.1 gezeigten sechs Phasen, die hier genauer betrachtet werden. Wie bereits in den Abschnitten 2.2.2–2.2.5 beschrieben, können verschiedene Operatoren, beispielsweise für die Mutation der Lösungen, eingesetzt werden. Dessen Austauschbarkeit wird nun durch dessen Komponentisierung und Definition von Kombinatoren für CLS ermöglicht. Dafür wird jedem Variationspunkt ein Schnittstellentyp zugeordnet. In der Python-Implementierung geschieht dies durch Definition eines abstrakten Kombinators. Instanzen der Klasse können mittels Subtyprelation inhabitiert werden:

- **Initialisierung** → *SolutionGenerator*
- **Elternselektion** → *ParentsSelectionOperator*
- **Rekombination** → *CrossoverOperator*
- **Mutation** → *MutationOperator*
- **Überlebensselektion** → *SurvivalSelectionOperator*

Für jeden Operator wurden in dieser Arbeit exemplarisch zwei verschiedene Verfahren implementiert, die sich an den Methoden, die in Abschnitt 2.2 erläutert wurden, orientieren. Bei den Selektionsmethoden kann die Turnierselektion für beide Selektionsschritte verwendet werden. Eine Übersicht ist in Abbildung 13 als Baumstruktur in Anlehnung an die Darstellung aus Winkels [68, S. 79] zu sehen. Die Operatoren werden im Folgenden einzeln erläutert. Anschließend erfolgt in Abschnitt 4.2.6 die Zusammenführung zu einem lauffähigen Algorithmus. In Abschnitt 4.2.7 wird der Aufbau eines Luigi-Tasks und die Verknüpfung zwischen den Algorithmen und Kombinatoren näher beschrieben und in Abschnitt 4.3 wird anschließend die automatisierte Suche und Kombination der verschiedenen Kombinatoren beschrieben.

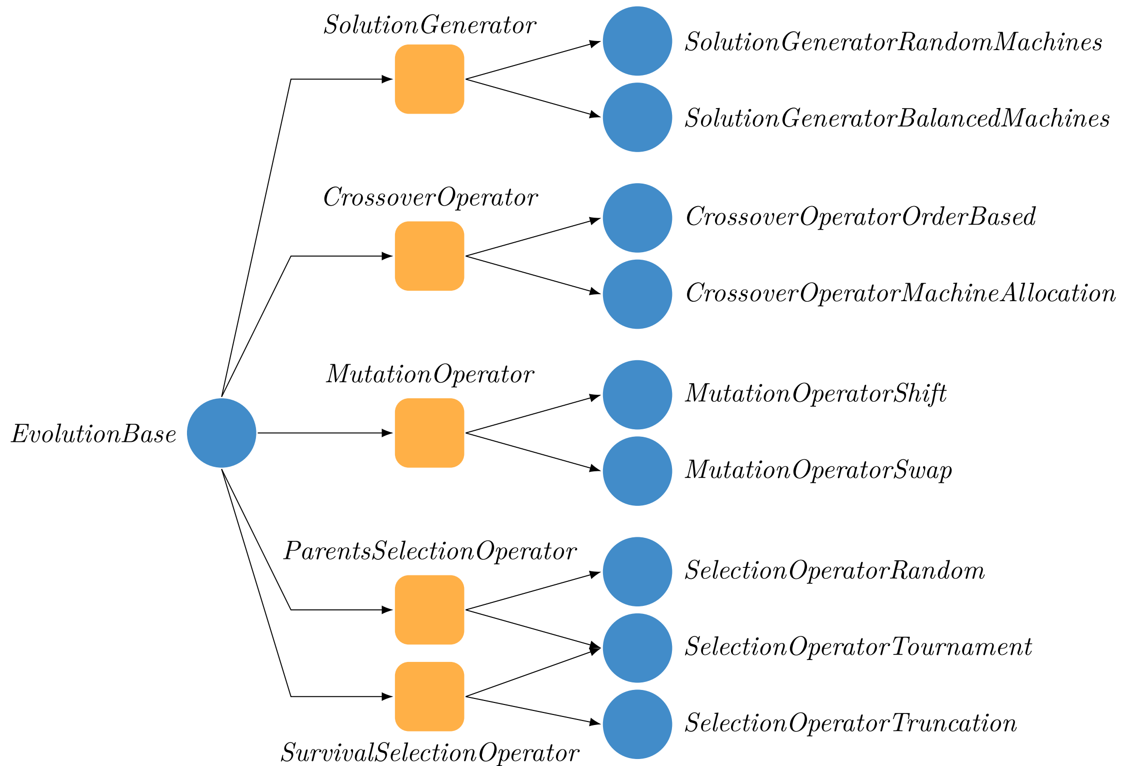


Abbildung 13: Übersicht über die implementierten Operatoren des Evolutionären Algorithmus.

4.2.1 Generatoren für die Initiallösungen

Der Evolutionäre Algorithmus benötigt initiale Lösungen entsprechend der Populationsgröße. Diese sollten möglichst gute Startwerte liefern, aber insbesondere den Suchraum möglichst gleichmäßig und unvoreingenommen abbilden, um eine hohe Diversität in der Population zu erzeugen. Dazu werden die Aufträge in beiden implementierten Operatoren gemischt und in zufälliger Reihenfolge auf die Maschinen verteilt.

Algorithmus 9 Lösungsgenerator zufällige Verteilung.

Require: Schedule s.

1: $jobs \leftarrow s.get_jobs_on_stage(stage = 0)$
2: $jobs \leftarrow shuffle(jobs)$
3: **for** Auftrag $job \in jobs$ **do**
4: $\quad machines \leftarrow s.get_eligible_machines(stage = 0, job = job)$
5: $\quad machine \leftarrow random(machines)$
6: $\quad s.allocate_tight(stage = 0, machine = machine, job = job)$
7: **end for**
8: **return** s

Algorithmus 9 belegt jeden Auftrag der Liste auf eine zufällige qualifizierte Maschine, während Algorithmus 10 unter den verfügbaren Maschinen diejenige mit der minimalen Endzeit, also der frühesten Startzeit des Auftrages gemäß *Earliest Start Time (EST)*,

Algorithmus 10 Lösungsgenerator ausgeglichene Verteilung.

Require: Schedule s.

1: $jobs \leftarrow s.get_jobs_on_stage(stage = 0)$
2: $jobs \leftarrow shuffle(jobs)$
3: **for** Auftrag $job \in jobs$ **do**
4: $machines \leftarrow s.get_eligible_machines(stage = 0, job = job)$
5: $machine \leftarrow$ Maschine in $machines$ mit minimaler Endzeit
6: $s.allocate_tight(stage = 0, machine = machine, job = job)$
7: **end for**
8: **return** s

auswählt. Durch die initiale Mischung der Auftragsreihenfolge unterliegt der Algorithmus weiterhin einer Zufälligkeit und erzeugt mit jedem Durchlauf unterschiedliche Ergebnisse. Im direkten Vergleich erzeugt Algorithmus 10 einen ausgeglicheneren Ablaufplan und tendenziell bessere Zielfunktionswerte, während Algorithmus 9 den gesamten Suchraum abbilden kann.

4.2.2 Eltern- und Überlebensselektion

Für den Selektionsschritt wurden drei Verfahren implementiert. Die zufällige Wahl aus Algorithmus 11 zieht n unterschiedliche zufällige Individuen aus der Population, äquivalent zum in Alg. 1 beschriebenen Verfahren ohne Zurücklegen. Sie kann für die Elternselektion genutzt werden.

Algorithmus 11 Selektionsoperator zufällige Wahl.

Require: Population pop, Zielgröße n.

1: $new_pop \leftarrow$ Ziehe n Individuen aus Population pop ohne Zurücklegen.
2: **return** new_pop

Algorithmus 12 sortiert die Individuen aufsteigend nach ihren Fitnesswerten und gibt die besten n zurück. Er ist äquivalent zur $(\mu+\lambda)$-Selektion aus Algorithmus 4. Dieser kann für die Überlebensselektion verwendet werden. In jeder Iteration werden die schlechtesten Individuen aus der Population verdrängt.

Algorithmus 12 Selektionsoperator elitistische Wahl.

Require: Population pop, Zielfunktion obj, Zielgröße n.

1: $sorted_pop \leftarrow$ Sortiere Population pop Aufsteigend nach Zielfunktionswert gemäß obj.
2: **return** $sorted_pop[0 : n]$

Algorithmus 13 implementiert eine Turnierselektion äquivalent zu Algorithmus 5. In jedem Schritt wird ein Individuum aus der Population ausgewählt. Dazu werden Gruppen der Größe $\kappa * n$ mit Zurücklegen aus der Population gezogen. Das Individuum mit dem kleinsten Zielfunktionswert wird zur neuen Population hinzugefügt. Existieren meh-

rere Individuen mit dem gleichen Zielfunktionswert, wird zufällig gewählt. Die Größe der Gruppen steuert wie in Abschnitt 2.2.2 beschrieben den Selektionsdruck. In der Implementierung aus Algorithmus 13 wird dazu der Parameter κ mit $0 < \kappa < \infty$ als Anteil der Populationsgröße gewählt. Die Gruppen können demnach durch das Zurücklegen auch größer als die Populationsgröße werden. Für $\kappa \to 0$ wird der Algorithmus ähnlich zur zufälligen Wahl aus Algorithmus 11, allerdings mit der Möglichkeit Individuen mehrfach zu wählen. Für $\kappa \to \infty$ geht die Wahrscheinlichkeit, dass das beste Individuum nicht in der Gruppe ist, gegen null, sodass die neue Population ausschließlich aus Individuen mit dem besten Zielfunktionswert besteht. In den Auswertungen in Abschnitt 5 wird $\kappa = 0.1$ gesetzt.

Algorithmus 13 Selektionsoperator Turnierselektion.

Require: Population *pop*, Zielfunktion *obj*, Zielgröße n.

1: $new_pop \leftarrow \varnothing$
2: **repeat** n
3: $\quad group \leftarrow \varnothing$
4: $\quad$ **repeat** $\lceil \kappa * n \rceil$
5: $\qquad group \leftarrow group \cup random(pop)$
6: $\quad$ **end**
7: $\quad next \leftarrow$ Individuum aus *group* mit kleinstem Zielfunktionswert gemäß *obj*
8: $\quad new_pop \leftarrow new_pop \cup next$
9: **end**
10: **return** new_pop

Die Turnierselektion kann sowohl für die Elternselektion, als auch für die Überlebensselektion verwendet werden. Bei der Überlebensselektion ist es möglich, dass das beste Individuum nicht überlebt. Im Framework wird daher zusätzlich Elitismus wie in Abschnitt 2.2.2 unter Algorithmus 3 implementiert. Das Framework wird genauer in Abschnitt 4.2.6 beschrieben.

4.2.3 Rekombination

Die Rekombinationsoperatoren erzeugen aus zwei Individuen der Population einen neuen Nachkommen. Es wurden zwei Operatoren entwickelt. Da durch die Trennung der Aufträge auf verschiedene Maschinen kein einzelnes Chromosom mehr konstruierbar ist, ohne die unkontrollierte Verschiebung von Aufträgen auf potenziell nicht qualifizierte Maschinen zu ermöglichen, wurden für die beiden Algorithmen Einschränkungen an die Wahl der Crossoverpunkte getroffen. Alternativ könnten neue Lösungen auf Zulässigkeit geprüft und unzulässige Lösungen verworfen werden. Da die Crossoveroperatoren jedoch mitunter sehr große Änderungen am Chromosom vornehmen und bereits eine Zuweisung eines Auftrages auf eine nicht qualifizierte Maschine die Lösung unzulässig macht, ist die zu erwartende Rate an unzulässigen Lösungen sehr hoch und damit das Verfahren ineffizi-

Algorithmus 14 Rekombinationsoperator Order-Based.

Require: Eltern s_1, s_2.

1: Hole Maschinenzuweisungen π_1, π_2 auf zufälliger Maschine *machine* aus Eltern s_1, s_2.

2: Bestimme Crossoverpunkte c_1 und c_2 mit $0 \leq c_1 \leq c_2 < |\pi_1|$.

3: Kopiere Gene aus π_1 zwischen c_1 und c_2 nach π indexgleich.

4: Kopiere Gene aus π_2, die noch nicht in π enthalten sind, beginnend bei c_2.

5: Kopiere restliche Gene aus π_1, die noch nicht in π enthalten sind, beginnend bei aktueller Position.

6: Entferne alle Maschinenzuweisungen in s für Maschine *machine* und plane neues Chromosom ein.

7: **return** s

enter als der zusätzliche Schritt der Sucheinschränkung. Algorithmus 14 basiert auf dem in Abschnitt 2.2.5 unter Algorithmus 8 vorgestellten *Order-Based-Crossover*. Er wird hier ausschließlich auf die Aufträge einer zufälligen Maschine angewendet. Alle anderen Maschinenzuweisungen werden aus dem ersten Elter übernommen. Innerhalb der gewählten Maschine werden die Crossoverpunkte c_1 und c_2 mit $c_1 \leq c_2$ bestimmt. Alle Maschinenzuweisungen aus dem ersten Elter werden im Intervall $[c_1, c_2)$ in das Nachkommenchromosom kopiert. Anschließend werden alle Aufträge aus dem zweiten Elter, die noch nicht eingeplant wurden, beginnend bei c_2, eingeplant. Wird dabei die Array-Grenze erreicht, wird am Anfang fortgefahren. Da die Menge der auf der gewählten Maschine eingeplanten Aufträge in den beiden Eltern nicht identisch sein muss, werden nur Aufträge berücksichtigt, die auch im ersten Elter vorhanden sind. Dies vermeidet Dopplungen von Aufträgen auf anderen Maschinen. Abschließend müssen alle Aufträge des ersten Elters, die noch nicht eingeplant wurden, weil sie nicht im zweiten Elter enthalten waren, eingeplant werden. Der Algorithmus verschiebt keine Aufträge auf andere Maschinen. Lediglich die Reihenfolge der Aufträge auf einer zufälligen Maschine wird basierend auf der Reihenfolge eines anderen Individuums verändert. Der vollständige Algorithmus befindet sich im Anhang unter Alg. 23.

Zusätzlich zum *Order-Based-Crossover* wurde ein Rekombinationsoperator zur Mischung der Maschinenzuweisungen entworfen. Dieser behält ebenfalls die relative Reihenfolge der Aufträge bei und rekombiniert die Zuweisungen auf den Maschinen. Der Algorithmus ist in Alg. 15 dargestellt, eine vollständige Darstellung befindet sich im Anhang unter Alg. 24. Im ersten Schritt wird ein Crossoverpunkt c gewählt, der die Maschinen in zwei Gruppen aufteilt. Aus dem ersten Elter werden nun alle Zuweisungen auf den Maschinen $< c$ übernommen. Die Zuweisungen auf den Maschinen $\geq c$ werden ohne Duplikate der bereits eingeplanten Aufträge aus dem zweiten Elter übernommen. Abschließend werden die verbliebenen, noch nicht eingeplanten Aufträge, aus den Eltern kopiert. Dazu wird das

Algorithmus 15 Rekombinationsoperator Maschinenzuweisungen.

Require: Eltern s_1, s_2.

1: $s \leftarrow s_1.shallow_copy()$
2: $num_machines \leftarrow schedule.get_machines_on_stage(stage = 0)$
3: ▷ Bestimme Crossoverpunkt
4: $c \leftarrow random(0, |s_1.get_machines_on_stage(stage = 0)|)$
5: $planned \leftarrow \varnothing$
6: Kopiere Maschinenzuweisungen aus s_1 bis Crossoverpunkt c im Intervall [0, c) und speichere sie in *planned*.
7: Kopiere Maschinenzuweisungen aus s_2, die nicht in *planned* enthalten sind, ab Crossoverpunkt c im Intervall [c, num_machines) und füge sie zu *planned* hinzu.
8: Kopiere Maschinenzuweisungen aus s_1, die nicht in *planned* enthalten sind, bis Crossoverpunkt c im Intervall [0, c) und füge sie zu *planned* hinzu.
9: Kopiere Maschinenzuweisungen aus s_2, die nicht in *planned* enthalten sind, ab Crossoverpunkt c im Intervall [c, num_machines).
10: **return** s

jeweils komplementäre Maschinenintervall auf den Eltern betrachtet und alle noch nicht eingeplanten Aufträge an die entsprechenden Zuweisungslisten angehangen.

4.2.4 Mutation

Die implementierten Mutationsoperatoren *Shift* und *Swap* ändern bestehende Individuen. Sie wurden analog zu den in Abschnitt 2.2.4 vorgestellten Verfahren *Translokation* (Alg. 7) und *2-Swap* (Alg. 6) entwickelt.

Der Mutationsoperator *Shift* aus Algorithmus 16 wählt zunächst einen zufälligen Auftrag aus der Auftragsmenge der ersten Bearbeitungsstufe aus. Dieser wird an eine zufällige Position auf einer zufälligen qualifizierten Maschine verschoben. Es ist zu beachten, dass der Mutationsoperator *Shift* nach Auswahl eines zufälligen Auftrages immer eine gültige Position findet. Im Extremfall kann dies jedoch auch die aktuelle Position sein, wodurch das Chromosom faktisch unverändert bleibt. Eine vollständige Darstellung befindet sich im Anhang unter Alg. 25.

Der Mutationsoperator *Swap* aus Algorithmus 17 wählt im ersten Schritt ebenfalls einen zufälligen Auftrag aus. Anschließend wird aus der Menge der verbliebenen Aufträge ein zufälliger Auftrag gewählt. Dabei muss jeweils die aktuelle Maschine des Auftrages für den jeweils anderen qualifiziert sein. Existiert kein solcher Auftrag, wird das Individuum unverändert zurückgegeben. Falls ein Paar gefunden wurde, werden die Maschinenzuweisungen getauscht, das heißt die Aufträge werden auf der jeweils anderen Maschine an der ursprünglichen Stelle eingeplant. Der vollständige Algorithmus befindet sich im Anhang unter Alg. 26.

Algorithmus 16 Mutationsoperator Shift.

Require: Schedule s.
1: $jobs \leftarrow s.get_scheduled_jobs_on_stage(stage = 0)$
2: **if** $|jobs| == 0$ **then**
3: **return** s
4: **end if**
5: Wähle einen zufälligen Auftrag job aus $jobs$.
6: Wähle eine neue Maschine aus allen qualifizierten Maschinen.
7: Entferne Auftrag job von ursprünglicher Maschine.
8: Füge Auftrag job an einer zufälligen Position auf neuer Maschine ein.
9: Korrigiere Start- und Endzeiten aller Aufträge auf beiden Maschinen.
10: **return** s

Algorithmus 17 Mutationsoperator Swap.

Require: Schedule s.
1: $jobs \leftarrow s.get_scheduled_jobs_on_stage(stage = 0)$
2: **if** $|jobs| \leq 1$ **then**
3: **return** s
4: **end if**
5: $jobs \leftarrow shuffle(jobs)$
6: $j_1, m_1, index_1 \leftarrow jobs[0]$
7: **for** $j_2, m_2, index_2 \in jobs[1:]$ **do**
8: ▷ Prüfe gegenseitige Maschinenqualifikation
9: **if** $\neg e_{0m_1j_2} \vee \neg e_{0m_2j_1}$ **then**
10: **continue**
11: **end if**
12: Tausche die Aufträge j_1 und j_2.
13: Korrigiere Start- und Endzeiten aller Aufträge auf beiden Maschinen.
14: **return** s
15: **end for**
16: **return** s

Vergleicht man die beiden Operatoren bezüglich Konvergenzgeschwindigkeit und Suchraum (Erreichbarkeit), lässt sich feststellen, dass der Operator *Swap* die Lösung mit doppelter Geschwindigkeit verändert. Jede Vertauschung von Aufträgen lässt sich mittels zwei Translokationen ebenfalls erwirken. Durch den paarweisen Tausch bildet der *Swap*-Operator jedoch einen kleineren Suchraum ab. Insbesondere ändert er niemals die Anzahl der eingeplanten Aufträge auf einer Maschine. In Kombination mit dem Rekombinationsoperator Order-Based aus Algorithmus 14 bleibt die Anzahl der eingeplanten Aufträge auf den Maschinen identisch zur Startlösung. Im schlechtesten Fall erzeugt die zufällige Zuweisung ein Chromosom, das alle Aufträge ausschließlich auf einer Maschine zugewiesen hat, was sich bis zum Ende der Suche nicht ändern kann. Der Mutationsoperator *Shift*

hingegen bildet den gesamten Suchraum ab. Jede valide Konfiguration von Maschinenzuweisungen lässt sich mit einer endlichen Anzahl an Translokationen aus jeder gültigen Startkonfiguration erreichen.

4.2.5 Stufenstrategie

Alle bislang vorgestellten Operatoren arbeiten ausschließlich auf der ersten Bearbeitungsstufe. Um den Algorithmus nun vom parallelen Maschinenproblem auf einen hybriden Flow Shop zu erweitern, wird eine Stufenstrategie hinzugenommen, die auf Basis der ersten Bearbeitungsstufe alle weiteren Stufen einplant. Die Stufenstrategie ist ein Parameter des Algorithmus und somit austauschbar, in dieser Arbeit kommt zunächst jedoch nur die Strategie *Earliest completion time (ECT)* zum Einsatz, da sie in der Literatur am häufigsten für (hybride) Flow Shops angewendet wird [49, 60]. Diese plant die Aufträge in derjenigen Reihenfolge ein, dass iterativ immer derjenige Auftrag gewählt und platziert wird, der seine Bearbeitung zuerst abschließen kann. Der Algorithmus ist in Alg. 18 gezeigt.

Algorithmus 18 Stufenstrategie Earliest Completion Time (ECT).

Require: Schedule s, Stufe i.

1: Entferne alle Maschinenzuweisungen auf Stufe i.

2: $jobs \leftarrow \{j \mid j \in s.jobs \land \neg skip_{ij}\}$
3: **while** $|jobs| > 0$ **do**
4: $\quad ects \leftarrow []$
5: $\quad$ **for** $job \in jobs$ **do**
6: $\quad\quad$ **for** Machine $l \in \{l \mid l \in M_i \land E_{ilj}\}$ **do**
7: $\quad\quad\quad ect \leftarrow max(job.end_time(), machine.end_time()) + p_{ilj}$
8: $\quad\quad\quad ects \leftarrow ects \cup (job, machine, ect)$
9: $\quad\quad$ **end for**
10: $\quad$ **end for**

11: $\quad$ Sortiere $ects$ nach drittem Wert im Tupel (ECT).
12: $\quad job, machine, ect \leftarrow ects[0]$

13: $\quad$ Füge Auftrag job auf Maschine $machine$ mit Endzeit ect ein.
14: $\quad jobs \leftarrow jobs \setminus job$
15: **end while**

16: **return** s

4.2.6 Framework

Der Framework-Algorithmus führt die einzelnen Operatoren, die in den vorangegangenen Abschnitten erläutert wurden, zu einem lauffähigen Gesamtsystem zusammen. Der zugehörige Kombinator dient ebenfalls als Ziel der Syntheseanfrage und hat den semantischen Typen

$$SolutionGeneratorCombinator \rightarrow CrossoverOperatorCombinator \\ \rightarrow MutationOperatorCombinator \rightarrow ParentsSelectionOperatorCombinator \\ \rightarrow SurvivalSelectionOperatorcombinator \rightarrow \mathbf{EvolutionBaseCombinator}.$$

Der Algorithmus ist in Alg. 19 gezeigt und orientiert sich an dem allgemeinen Aufbau eines Evolutionären Algorithmus, der in Abschnitt 2.2 in Abbildung 4 beschrieben wurde. Im ersten Schritt wird eine Population der Größe *pop_size* generiert. Dazu wird der *solution_generator* aufgerufen, der als Eingabe den leeren Ablaufplan entgegennimmt und die Aufträge aus der Prozesszeitenmatrix unter Berücksichtigung der Nebenbedingungen auf die Maschinen verteilt. Anschließend wird das beste Individuum in *best* gespeichert.

In der folgenden Schleife werden die Operatoren auf die Population angewendet. Die Abbruchbedingung ist wählbar zwischen Laufzeit, Schleifeniterationen und Anzahl Zielfunktionsauswertungen. Zunächst werden aus der aktuellen Population *par_size*-viele Eltern für die Rekombination und Mutation ausgewählt. Paare aus je zwei Individuen werden mit der Wahrscheinlichkeit p_c durch den Operator *crossover_operator* rekombiniert. Die Eltern werden zunächst gemischt, um eine zufällige Paarung zu erreichen. Der entstandene Nachkomme wird einer neuen Menge pop_c der rekombinierten Individuen hinzugefügt. Die Individuen aus der Elternschaft und die neu generierten Nachkommen in pop_c werden anschließend jeweils mit der Wahrscheinlichkeit p_m durch den Operator *mutation_operator* mutiert und zur neuen Menge pop_m hinzugefügt.

Die neue Population ergibt sich aus der Vereinigung der alten Population mit den rekombinierten und mutierten Individuen aus pop_c und pop_m. Anschließend wird die Überlebensselektion durch den Operator *survival_selection_operator* durchgeführt. Da bei der Überlebensselektion je nach Wahl des Operators das beste Individuum verloren gehen kann, wird anschließend Elitismus angewendet, der in Abschnitt 2.2.2 erläutert wurde. Dabei wird im Fall, dass das beste Individuum der Population einen schlechteren Zielfunktionswert aufweist als das in *best* gespeicherte, das schlechteste durch *best* ersetzt. Abschließend wird das aktuell beste Individuum aktualisiert.

Algorithmus 19 Basisalgorithmus.

```
Require: Schedule s, Zielfunktion obj.
 1: ▷ Initiale Population generieren
 2: pop ← ∅
 3: repeat pop_size
 4:     sol ← solution_generator.execute(s)
 5:     pop ← pop ∪ stage_allocation_strategy.execute(sol, 1)
 6: end
 7: best ← Individuum aus pop mit kleinstem Zielfunktionswert gemäß obj
 8: while ¬STOP do
 9:     ▷ Parent selection
10:     parents ← parents_selection_operator.execute(pop, obj, par_size)
11:     ▷ Crossover
12:     pop_c ← ∅
13:     parents ← shuffle(parents)
14:     for s_1, s_2 ∈ zip(parents[0 :: 2], parents[1 :: 2]) do        ▷ Zip 2er-Paare
15:         if random(0, 1) ≤ p_c then
16:             sol ← crossover_operator.execute(s_1, s_2)
17:             pop_c ← pop_c ∪ stage_allocation_strategy.execute(sol, 1)
18:         end if
19:     end for
20:     ▷ Mutation
21:     pop_m ← ∅
22:     for s ∈ parents ∪ pop_c do
23:         if random(0, 1) ≤ p_m then
24:             sol ← mutation_operator.execute(s)
25:             pop_m ← pop_m ∪ stage_allocation_strategy.execute(sol, 1)
26:         end if
27:     end for
28:     ▷ Überlebensselektion
29:     pop ← pop ∪ pop_c ∪ pop_m
30:     pop ← survival_selection_operator.execute(pop, obj, pop_size)
31:     ▷ Elitismus
32:     new_best ← Individuum aus pop mit kleinstem Zielfunktionswert gemäß obj
33:     if C_max(new_best) > C_max(best) then
34:         worst_sol ← Individuum aus pop mit größtem Zielfunktionswert gemäß obj
35:         pop ← (pop \ worst_sol) ∪ best_sol
36:     end if
37:     ▷ Update
38:     if C_max(new_best) < C_max(best) then
39:         best ← new_best
40:     end if
41: end while
42: return best_solution
```

4.2.7 Verknüpfung von Algorithmen und Kombinatoren

Für jeden Operator, sowie den Framework-Algorithmus wird ein Kombinator definiert, um eine Inhabitation durch CLS zu ermöglichen. Als Namenskonvention erhält der Kombinator den Suffix *Combinator* an den Klassennamen:

$$\begin{aligned} \textit{MutationOperatorShift} &\longleftrightarrow \textit{MutationOperatorShiftCombinator} \\ \textit{MutationOperatorSwap} &\longleftrightarrow \textit{MutationOperatorSwapCombinator} \\ &\vdots \end{aligned}$$

Jeder Kombinator implementiert einen Luigi-Task und definiert, falls vorhanden, alle weiteren Voraussetzungen als Felder der Klasse. CLS generiert aus der Menge der Voraussetzungen automatisch einen Pfeiltyp. Für Operatoren ohne Voraussetzungen ergibt sich ihr Typ aus der abstrakten Oberklasse. Für die beiden Mutationsoperatoren aus dem vorherigen Beispiel ist dies der Typ *MutationOperatorCombinator*. Erwartet eine übergeordnete Komponente, wie der Framework-Algorithmus, nun einen Mutationsoperator, so erfüllen beide Kombinatoren *MutationOperatorShiftCombinator* und *MutationOperatorSwapCombinator* über die Subtyprelation (7) die Voraussetzung:

$$MutationOperatorCombinator \geq MutationOperatorShiftCombinator$$
$$MutationOperatorCombinator \geq MutationOperatorSwapCombinator$$

Bei den Selektionsoperatoren stehen drei Operatoren zur Verfügung. Die zufällige Selektion kann für die Elternselektion, die Truncation-Selektion für die Überlebensselektion eingesetzt werden:

$$ParentsSelectionOperatorCombinator \geq SelectionOperatorRandomCombinator$$
$$SurvivalSelectionOperatorCombinator \geq SelectionOperatorTruncationCombinator.$$

Für die Turnierselektion, die sowohl für die Eltern-, als auch für die Überlebensselektion verwendet werden kann, wird ein Intersektionstyp verwendet:

$$\begin{aligned} &SelectionOperatorTruncationCombinator : \\ &\quad SurvivalSelectionOperatorCombinator \cap ParentsSelectionOperatorCombinator \end{aligned}$$

Der Kombinator für den Framework-Algorithmus ist in Abbildung 15 dargestellt. Er enthält, anders als die Kombinatoren der Operatoren, die Definition der Voraussetzungen, aus denen sich der Pfeiltyp ableitet. Jede Voraussetzung wird als *CLSParameter* in einem Feld gespeichert. CLS durchsucht die Kombinatordefinitionen zur Laufzeit nach diesen Feldern und befüllt sie mit entsprechenden Inhabitanten des definierten Zieltyps. Der Typ des Kombinators ergibt sich in Abb. 14 schließlich als

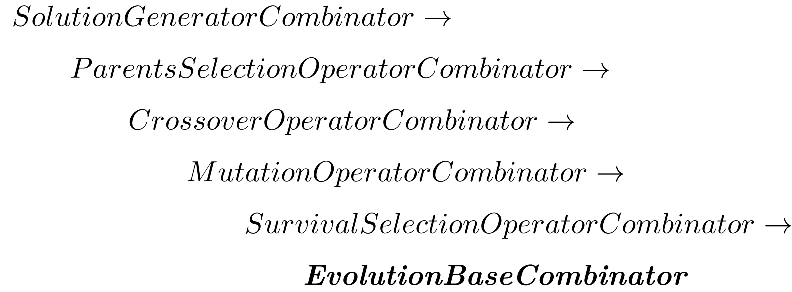


Abbildung 14: Semantischer Typ des Zielkombinators.

Darüber hinaus werden die numerischen Optimierungsparameter, die durch Luigi verarbeitet werden, definiert. Luigi verschiebt diese in die Parameterliste des Konstruktors, sodass sie bei der Instanziierung übergeben werden können. Bei den Parametern wurde eine Unterscheidung zwischen den bereits in Luigi bestehenden Typen (zum Beispiel *IntParameter*) und der neu eingeführten Familie der *OptimizationLuigiParameter* eingeführt. Diese stellen Optimierungsparameter in den Ausprägungen *bool*, *int* und *float* bereit, die später zur Optimierung anhand des Typen automatisiert erkannt werden können. Die vorhandenen Luigi-Parameter werden verwendet, um feste Werte wie die Anzahl der Replikationen an den Algorithmus übergeben zu können. Der *ScheduleTargetParameter* ist darüber hinaus ein Wrapper-Objekt für den Luigi Parameter für *Schedule*-Objekte. Hier können die Eingangsdaten an den Algorithmus übergeben werden und ein *Callback* für die Ergebnisse definiert werden. Der erste Block definiert die Anzahl der Replikationen, die der Algorithmus pro Instanz durchführen soll. Eine Mittlung der Ergebnisse findet hier noch nicht statt, es werden alle gefundenen Lösungen zurückgegeben. Der zweite Block definiert die Daten-Eingänge und -Ausgänge als *InMemoryTarget*. Das Feld *data* übergibt einen leeren Ablaufplan als *Schedule*-Objekt an den Algorithmus, der alle benötigten Daten für die Belegung enthält. Das *callback*-Feld wird vom Algorithmus mit den gefundenen Lösungen befüllt und kann im Anschluss an die Ausführung vom Framework ausgelesen werden. Durch die Nutzung vom Luigi-Framework ist es nicht möglich, Daten über (Konstruktor-) Parameter und Rückgabewerte auszutauschen, sodass dieses Vorgehen nötig wird.

Die folgenden Parameter definieren die numerischen und kategorischen Optimierungsparameter. Bei den numerischen Parametern kann die Populationsgröße *pop_size*, die Anzahl der gewählten Eltern pro Generation *par_size* prozentual in Abhängigkeit von der Populationsgröße, die Crossoverwahrscheinlichkeit p_c und die Mutationswahrscheinlichkeit p_m übergeben werden. Bei den Optimierungsparametern kann im Algorithmus ein zulässiger Wertebereich angegeben werden. Für die Wahrscheinlichkeiten wurde der volle Wertebereich $[0, 1]$ erlaubt, für die Populationsgröße wurde ein Bereich zwischen einem und 100 Individuen festgelegt. Bei der Elterngröße muss sichergestellt sein, dass mindestens

EvolutionBaseCombinator
Parameters + num_replications : IntParameter(*default*=20) # Data input and output requirements + data = ScheduleTargetParameter(*default*=InMemoryTarget()) + callback = ScheduleTargetParameter(*default*=InMemoryTarget()) # Numeric optimization parameters + pop_size = OptimizationLuigiParameter(*default*=IntegerOptimizationParameter(*lb*=1, *ub*=100)) + par_size = OptimizationLuigiParameter(*default*=FloatOptimizationParameter(*lb*=0.1, *ub*=1)) + p_c = OptimizationLuigiParameter(*default*=FloatOptimizationParameter(*lb*=0, *ub*=1)) + p_m = OptimizationLuigiParameter(*default*=FloatOptimizationParameter(*lb*=0, *ub*=1)) # Combinator optimization parameters + solution_generator = ClsParameter(*tpe*=SolutionGeneratorCombinator.return_type()) + parents_selection_operator = ClsParameter(*tpe*=ParentsSelectionOperatorCombinator.return_type()) + crossover_operator = ClsParameter(*tpe*=CrossoverOperatorCombinator.return_type()) + mutation_operator = ClsParameter(*tpe*=MutationOperatorCombinator.return_type()) + survival_selection_operator = ClsParameter(*tpe*=SurvivalSelectionOperatorCombinator.return_type())
+ ___init___(self, *args, **kwargs) + run(self) + run_single(self) : Schedule + run_replicated(self) : list[Schedule]

Abbildung 15: Kombinator für den Basisalgorithmus.

ein Individuum ausgewählt wird, da der Algorithmus sonst keine Änderungen vornimmt. Der zulässige Wertebereich wurde daher auf $[0.1, 1]$ beschränkt. Für die kategorischen Parameter wird der semantische Typ des abstrakten Kombinators deklariert. Zur Laufzeit führt das Luigi-Framework den Task aus und CLS löst dabei die Voraussetzungen auf. Die Parameter *data* und *callback*, sowie die numerischen Parameter, werden Luigi dabei als Argumentliste übergeben und automatisch in den entsprechenden Feldern gesetzt. Die benötigten Kombinatoren werden im Konstruktor der Klasse als Voraussetzung für die Ausführung des Tasks angegeben, sodass Luigi automatisch deren Ausführung voranstellt. Alle Luigi-Tasks sind einheitlich implementiert, sodass sie als Ergebnis der Ausführung ihrer *run*-Methode eine Instanz des zugeordneten Algorithmus in ein *InMemoryTarget* speichern, das anschließend vom übergeordneten Kombinator genutzt werden kann. Diese wurden als Erweiterung der existierenden *Tasks* implementiert, um einen performanten Objektaustausch über den Arbeitsspeicher ohne eine Form von Serialisierung zu ermöglichen. In Algorithmus 20 ist die Verwendung des *Targets* als Ergebnis der Ausführung des Luigi-Tasks am Beispiel des Mutationsoperators *MutationOperatorShift* mit seinem zugeordneten Kombinator *MutationOperatorShiftCombinator* verdeutlicht.

Algorithmus 20 Klasse MutationShiftCombinator.

```
1:  class MutationOperatorShiftCombinator(MutationOperatorCombinator):
2:      abstract = False
3:
4:      def __init__(self, *args, **kwargs):
5:          super().__init__(name=„Mutation operator Swap“, *args, **kwargs)
6:          self.target = InMemoryTarget()
7:          self.require([])
8:
9:      def run(self):
10:         self.target.save(MutationOperatorShift())
```

Er besitzt keine Voraussetzungen und übergibt Luigi daher per *require()* eine leere Liste. Als Austauschobjekt wird in *target* das *InMemoryTarget* gespeichert, das nach der Ausführung die Instanz des Operators beinhaltet. In der Frameworkimplementierung sind die Leer-Instanziierungen im Konstruktor und weitere Parameter zur Steuerung von Instanziierbarkeit und Replizierbarkeit in einer Vererbungshierarchie für den Anwender versteckt, sodass nur die Task-spezifischen Einstellungen getroffen werden müssen. Die Methode *run*, die von Luigi automatisch ausgeführt wird, speichert im *target* nun eine Instanz des implementierten Mutationsoperators. Im übergeordneten Task kann via *input()* anschließend darauf zugegriffen werden. Ein Auszug aus der *run*-Methode des Kombinators des Framework-Algorithmus ist in Alg. 21 gezeigt. Hier wird über *self.input()* auf die Ergebnisse der vorausgesetzten Tasks zugegriffen. Dafür wurde eine Struktur mittels

Algorithmus 21 run-Methode der Klasse EvolutionBaseCombinator.

```
1: def run(self):
2:     req: dict[ClsParameter, InMemoryTarget] = self.input()
3:     self.target.save(EvolutionBase(
4:         solution_generator=req[self.solution_generator].read(),
5:         parents_selection_operator=req[self.parents_selection_operator].read(),
6:         crossover_operator=req[self.crossover_operator].read(),
7:         mutation_operator=req[self.mutation_operator].read(),
8:         survival_selection_operator=req[self.survival_selection_operator].read(),
9:         pop_size=self.pop_size,
10:        par_size=self.par_size,
11:        p_c=self.p_c,
12:        p_m=self.p_m,
13:        stage_allocation_strategy=ECT(),
14:        objective=CMax(),
15:        stop_criterion=StopCriterionEvaluations(1000)
       ).execute(schedule=self.data.read().copy())
```

eines *dict* verwendet, sodass die Deklaration der Voraussetzungen auf die Instanzen der Algorithmenkomponenten gemappt werden kann. In den Zeilen 4 - 8 in Algorithmus 21 wird so auf die durch CLS kombinierten Algorithmenkomponenten bei der Instanziierung des Evolutionären Algorithmus zugegriffen. In den Zeilen 9 - 12 werden die durch Luigi gesetzten numerischen Parameter übergeben.

Diese Vorgehensweise der Trennung von Algorithmenimplementierung und Kombinatoren ermöglicht es, die Algorithmenkomposition vor der Ausführung zusammenzusetzen und anschließend auf die gegebenen Daten (ggf. mehrfach) anzuwenden, anstatt große Datenmengen während der Ausführung von Luigi über die Pipelines durchzureichen. Weiterhin findet dadurch eine saubere Trennung zwischen purer Algorithmik und Syntheseframework statt. Die Algorithmen können weiterhin unabhängig von CLS manuell zusammengesetzt und (ggf. automatisiert) getestet werden. Dies ermöglicht die vollkommen unabhängige Implementierung der beiden Teile. Das Luigi-Framework fängt während der Ausführungen Fehler, die in den einzelnen Tasks auftreten können, ab und verwirft diese. Dadurch kann auch mit fehlerhaften Komponenten im Repository weitergearbeitet werden, indem diese automatisch eliminiert werden und die Ausführung dennoch fortgeführt wird.

4.3 Dynamische Komponentensuche

Das entwickelte Syntheseframework soll automatisch ein Repository aus implementierten Kombinatoren aufbauen, ohne diese manuell registrieren zu müssen. Dies reduziert den Aufwand zur Erweiterung des Algorithmus durch neue Komponenten und versteckt die Nutzung von CLS. Dafür soll das Framework eine gegebene Paketstruktur automatisiert nach anwendbaren Kombinatoren durchsuchen können. Für eine Syntheseanfrage muss dann lediglich der Zieltyp der Anfrage und das root-Element der Paketstruktur spezifiziert werden. Zusätzlich lässt sich in der vorliegenden Implementierung eine *Blacklist* definieren, um bestimmte Kombinatoren temporär aus der Betrachtung zu entfernen. CLS durchsucht rekursiv die vorliegenden Quelldateien nach Kombinatoren und löst den Zieltyp auf. Aus der Menge aller gefundenen Kombinatoren wird anschließend das Repository generiert. Eine schematische Vorgehensweise der Komponentensuche ist in Abb. 16 skizziert. Die einzelnen Schritte werden im Folgenden näher beschrieben.

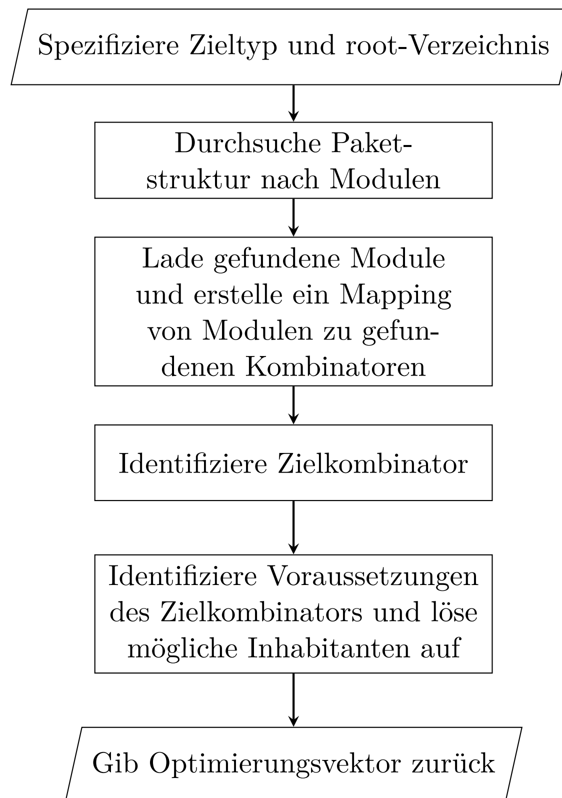


Abbildung 16: Schematische Vorgehensweise zur automatisierten Komponentensuche.

4.3.1 Modulsuche

Für die Parameteroptimierung wird zunächst der zu optimierende Algorithmus anhand des semantischen Typs ohne die Voraussetzungen des Kombinators übergeben. Für den hier implementierten Evolutionären Algorithmus lautet dieser *EvolutionBaseCombinator*. Die Voraussetzungen des Kombinators, die entsprechend den Pfeiltypen definieren, werden automatisch aufgelöst. Des Weiteren muss ein root-Verzeichnis für die Suche nach ladbaren Komponenten angegeben werden. In der konkreten Implementierung liegen alle implementierten Operatoren in *~\src\main\algorithms\ea*, sodass dieser Pfad als Startpunkt der rekursiven Verzeichnissuche gewählt werden kann. Der Algorithmus enumeriert zunächst alle Python-Module zur späteren Verwendung. Mittels einer *Blacklist* können einzelne Module temporär von der Betrachtung ausgenommen werden, um die Verwendung von einzelnen Operatoren steuern zu können. Des Weiteren werden alle *__init__.py* Dateien und das Verzeichnis *__pycache__* per Konvention ignoriert. Es resultiert eine Liste aller Python-Quelldateien mit ihrem Namespace-relativen import-Pfad, die im angegebenen Verzeichnis liegen. In der aktuellen Implementierung ergeben sich die in Tabelle 4 gezeigten 14 Module:

#	**Modulname**
1	src.main.algorithms.ea.base.evolution_base
2	src.main.algorithms.ea.crossover.crossover_operator
3	src.main.algorithms.ea.crossover.crossover_operator_machine_allocation
4	src.main.algorithms.ea.crossover.crossover_operator_order_based
5	src.main.algorithms.ea.mutation.mutation_operator
6	src.main.algorithms.ea.mutation.mutation_operator_shift
7	src.main.algorithms.ea.mutation.mutation_operator_swap
8	src.main.algorithms.ea.selection.selection_operator
9	src.main.algorithms.ea.selection.selection_operator_random
10	src.main.algorithms.ea.selection.selection_operator_tournament
11	src.main.algorithms.ea.selection.selection_operator_truncation
12	src.main.algorithms.ea.solutiongenerator.solution_generator
13	src.main.algorithms.ea.solutiongenerator.solution_generator_balanced_machines
14	src.main.algorithms.ea.solutiongenerator.solution_generator_random_machines

Tabelle 4: Übersicht der geladenen Module bei der Komponentensuche.

4.3.2 Erstellung eines Modulmappings

Im nächsten Schritt werden die gefundenen Module geladen und nach Kombinatoren durchsucht. Die CLS-Implementierung lässt kein manuelles Hinzufügen von Kombinatoren zu dem Repository zu. Stattdessen werden alle Klassen, die im Python-Interpreter importiert wurden, verwendet. Python verfügt über die Möglichkeit, Module während der Laufzeit dynamisch zum Namespace des Interpreters mittels *__import__()* aus dem *importlib*-Paket [88] hinzuzufügen. Die in einem Modul enthaltenen Kombinatoren können

also aus dem Vergleich der durch CLS generierten Repositorys vor und nach dessen Import ermittelt werden. Um Verluste durch Namensdopplungen zu vermeiden, wird sichergestellt, dass das Repository vor dem Import leer ist. Per Konvention darf jede Quelldatei nur maximal einen nicht-abstrakten Kombinator enthalten, um später das gezielte Hinzufügen eines Kombinators zum Repository zu ermöglichen. Nach dem Laden eines Moduls müssen die ggf. darin enthaltenen Klassen wieder aus dem Namespace entfernt werden. Da Python keine Möglichkeit des sicheren Entfernens von bereits importierten Klassen bereitstellt, muss der Zustand des Interpreters vor dem Import wiederhergestellt werden. Dazu wird für jedes zu ladende Modul der Prozess geforkt und dieser anschließend verworfen. Ein Ablaufdiagramm ist in Abbildung 17 gezeigt.

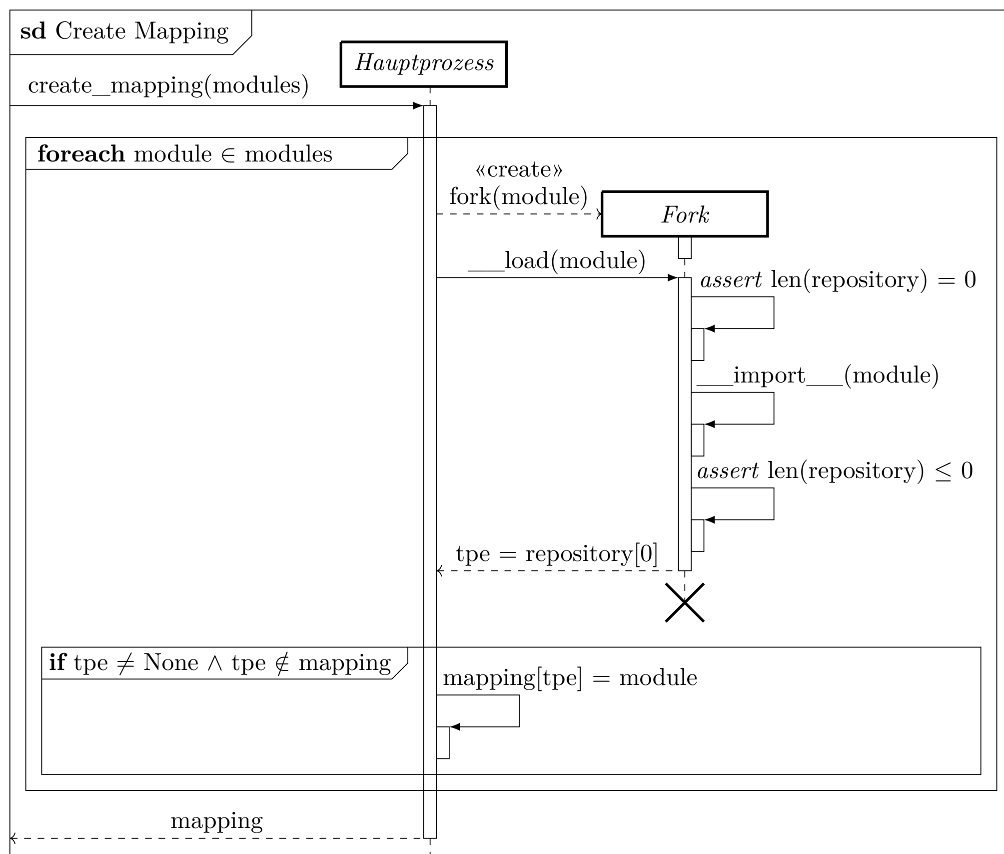


Abbildung 17: Sequenzdiagramm der Erstellung des Modulmappings.

Zu sehen ist, dass für jedes zu ladende Modul ein Fork erzeugt wird, der das Laden des Moduls in einer separierten Umgebung durchführt und das Ergebnis zurückliefert. Beim Fork wird der Zustand des aktuellen Interpreters geklont, sodass alle bereits durchgeführten Imports bestehen bleiben. Falls das geladene Modul genau einen Kombinator enthält, wird es zu einem *mapping*-Dictionary hinzugefügt, das am Ende der Methode zurückgegeben wird. Enthält ein Modul keinen Kombinator wird es ignoriert, enthält es mehrere, wird es übersprungen und eine Warnung wird ausgegeben. In der aktuellen Implementierung ergibt sich das in Tabelle 5 gezeigte Mapping:

#	Kombinator		Modul
1	EvolutionBaseCombinator	→	~.evolution_base
2	CrossoverOperatorMachineAllocationCombinator	→	~.crossover_operator_machine_allocation
3	CrossoverOperatorOrderBasedCombinator	→	~.crossover_operator_order_based
4	MutationOperatorShiftCombinator	→	~.mutation_operator_shift
5	MutationOperatorSwapCombinator	→	~.mutation_operator_swap
6	SelectionOperatorRandomCombinator	→	~.selection_operator_random
7	SelectionOperatorTournamentCombinator	→	~.selection_operator_tournament
8	SelectionOperatorTruncationCombinator	→	~.selection_operator_truncation
9	SolutionGeneratorBalancedMachinesCombinator	→	~.solution_generator_balanced_machines
10	SolutionGeneratorRandomMachinesCombinator	→	~.solution_generator_random_machines

Tabelle 5: Übersicht des Modulmappings bei der Komponentensuche.

Die Liste der zu betrachtenden Module hat sich auf diejenigen reduziert, die konkrete Implementierungen von genau einem Kombinator enthalten. Die vier Module

1. src.main.algorithms.ea.crossover.crossover_operator
2. src.main.algorithms.ea.mutation.mutation_operator
3. src.main.algorithms.ea.selection.selection_operator
4. src.main.algorithms.ea.solutiongenerator.solution_generator

mit den fünf abstrakten Kombinatoren

1. CrossoverOperatorCombinator
2. MutationOperatorCombinator
3. ParentsSelectionOperatorCombinator
4. SurvivalSelectionOperatorCombinator
5. SolutionGeneratorCombinator

wurden entsprechend entfernt. Diese dienen lediglich dem Subtyping (7) und werden inhärent durch die Subtypen mitgeladen. Aus der Liste der gefundenen Kombinatoren kann nun der gesuchte Zielkombinator abgeleitet werden. Konkret ist dies in der aktuellen Implementierung der Kombinator *EvolutionBaseCombinator* im Modul *~.evolution_base*. Zusätzlich zum in Tabelle 5 angegebenen nativen Typen des Kombinators wird auch der semantische (ggf. Pfeil-) Typ gespeichert. Für den geforderten Zielkombinator *EvolutionBaseCombinator* ergibt sich der schon in Abbildung 14 gezeigte Pfeiltyp

$$\begin{gathered} \mathit{SolutionGeneratorCombinator} \rightarrow \mathit{ParentsSelectionOperatorCombinator} \rightarrow \\ \mathit{CrossoverOperatorCombinator} \rightarrow \mathit{MutationOperatorCombinator} \rightarrow \\ \mathit{SurvivalSelectionOperatorCombinator} \rightarrow \boldsymbol{EvolutionBaseCombinator} \end{gathered}$$

aus dem sich im Folgenden die Voraussetzungen, das heißt insbesondere die austauschbaren Komponenten, ableiten lassen.

4.3.3 Identifizierung der Voraussetzungen des Zielkombinators

Um mögliche Inhabitanten für die fünf Voraussetzungen zu enumerieren, wird zunächst das gesamte Repository analog zum Schema aus Abbildung 17 isoliert vom Hauptprozess aufgebaut. Abweichend entfällt die Prüfung, dass genau ein Kombinator gefunden wurde. Anschließend wird für jede Voraussetzung eine Syntheseanfrage gestellt und die Ergebnisse aufgelistet. Es ergeben sich die in Tabelle 6 gezeigten Inhabitanten. Zusätzlich wird der Zielkombinator nach numerischen Parametern durchsucht. Dazu werden alle Felder der Klasse nach Instanzen von *OptimizationParameter* durchsucht. Die Vererbungsstruktur in *BoolOptimizationParameter*, *IntegerOptimizationParameter* und *FloatOptimizationParameter* ermöglicht die spätere Erzeugung des Werteraums. Dieser kann mittels der Parameter *lb* (*lower bound* - untere Schranke) und *ub* (*upper bound* - obere Schranke) wie bereits in Abschnitt 4.2.7 erläutert ggf. eingeschränkt werden.

MutationOperatorCombinator:		
1	MutationOperatorSwapCombinator	
2	MutationOperatorShiftCombinator	
CrossoverOperatorCombinator:		
1	CrossoverOperatorOrderBasedCombinator	
2	CrossoverOperatorMachineAllocationCombinator	
SolutionGeneratorCombinator:		
1	SolutionGeneratorRandomMachinesCombinator	
2	SolutionGeneratorBalancedMachinesCombinator	
ParentsSelectionOperatorCombinator:		
1	SelectionOperatorRandomCombinator	
2	SelectionOperatorTournamentCombinator	
SurvivalSelectionOperatorCombinator:		
1	SelectionOperatorTournamentCombinator	
2	SelectionOperatorTruncationCombinator	
Numerische Parameter:		
1	pop_size	IntegerOptimizationParameter(lb=1, ub=100)
2	par_size	FloatOptimizationParameter(lb=0.1, ub=1.0)
3	p_c	FloatOptimizationParameter(lb=0.0, ub=1.0)
4	p_m	FloatOptimizationParameter(lb=0.0, ub=1.0)

Tabelle 6: Übersicht der gefundenen numerischen Parameter und Inhabitanten für die Voraussetzungen des Zielkombinators.

4.3.4 Ausführung der Parameterkonfigurationen

Während der Parameteroptimierung werden systematisch Parameterkonfigurationen getestet. Diese werden entweder iterativ auf Grund vorheriger Beobachtungen berechnet oder der Suchraum wird systematisch durchsucht. In jedem Fall liefert der Parameteroptimierer in jedem Schritt eine zu testende Konfiguration. Der Parametervektor der möglichen Belegungen der kategorischen und numerischen Parameter leitet sich direkt aus den in Tabelle 6 gezeigten Gruppen der Kombinatoren und der numerischen Parameter mit ihren Wertebereichen ab. Liefert der Parameteroptimierer für den nächsten Suchschritt eine Konfiguration, muss der Algorithmus entsprechend der Anfrage zusammengesetzt und parametrisiert werden. Ein Beispiel für einen Parametervektor ist in Algorithmus 22 dargestellt. Der Vektor besteht aus den getrennten Bereichen für kategorische und numerische Parameter, da dessen Konfiguration sich in die Nutzung von CLS und Luigi aufteilt.

Algorithmus 22 Beispiel für einen Parametervektor.

```
 1: OptimizationVector {
 2:     target = „EvolutionBaseCombinator“
 3:     mapping = { ... }
 4:     combinatorRequirements = {
 5:         „MutationOperatorSwapCombinator“: „MutationOperatorShiftCombinator“,
 6:         „CrossoverOperatorCombinator“: „CrossoverOperatorOrderBasedCombinator“,
 7:         „SolutionGeneratorCombinator“: „SolutionGeneratorRandomMachinesCombinator“,
 8:         „ParentsSelectionOperatorCombinator“: „SelectionOperatorTournamentCombinator“,
 9:         „SurvivalSelectionOperatorCombinator“: „SelectionOperatorTruncationCombinator“
10:     }
11:     numericRequirements = {
12:         "pop_size" : 100,   "par_size" : 0.1,   "p_c" : 0.25,   "p_m" : 0.5
13:     }
14: }
```

Für die Rekombination der Kombinatoren wird das Repository so aufgebaut, dass CLS genau die eine geforderte Konfiguration generiert. Dazu werden die geforderten Kombinatoren im Modulmapping auf die implementierenden Python-Module abgebildet und diese erneut analog zu dem in Abbildung 17 gezeigten Schema isoliert vom Hauptprozess geladen. Die numerischen Parameter können Luigi als Argumente in Form eines *dict* übergeben werden. Luigi erzeugt aus den Parameter-Feldern der Klasse zur Laufzeit Konstruktorparameter, die durch das *dict* befüllt werden. Dadurch, dass die Konfiguration der numerischen Parameter Teil der Synthese durch Luigi ist, kann nicht die Menge aller möglichen Konfigurationen der Ausführung vorangestellt werden. Es folgt eine Instanz des Zielkombinators, der entsprechend der vom Parameteroptimierer vorgegebenen Parameterkonfiguration erzeugt wurde. Die Ausführung des Luigi-Tasks resultiert wie in Abschnitt 4.2.7 beschrieben in einer Instanz des zugeordneten Algorithmus, der anschließend ausgewertet werden kann. Das Ergebnis der Ausführung wird dem Parameteroptimierer zurückgegeben.

4.4 Mathematisches Optimierungsmodell

Um die Ergebnisse des Evolutionären Algorithmus einordnen zu können, wurde eine ILP-Formulierung des Problems entwickelt und mittels eines Solvers für die gewählten Testinstanzen bis zur Optimalität gelöst. Es ist somit möglich, die Ergebnisse der heuristischen Läufe ins Verhältnis mit der optimalen Lösung zu setzen. Die Definition des ILP erfolgt in Anlehnung an das Modell von Ruiz et al. [13]. Dieses modelliert einen hybriden Flow Shop, der in Graham-Notation wie folgt angegeben werden kann [13, S. 5]:

$$FHm, ((RM^i)_{i=1}^m) \mid skip, M_j, r_m, prec, lag, S_{iljk}, A_{iljk} \mid C_{max}$$

Um die Komplexität des Modells und damit auch die Berechnungsdauer zu reduzieren, wurde das Modell an die gegebene Problemformulierung (1) angepasst, indem alle nicht benötigten Nebenbedingungen entfernt und die Übernommenen angepasst wurden. Im Folgenden wird nun zunächst auf die verwendeten Variablen und die Formulierung des ILP eingegangen.

4.4.1 Definition der Parameter und Variablen

Die Variablen wurden weitestgehend analog zu denen im Modell von Ruiz et al. [13] benannt. Die Entscheidungsvariablen, die vom Solver zu bestimmen sind, sind in Tabelle 7 aufgeführt. Diese stellen die Zuweisung und zeitliche Einplanung der Aufträge dar. Aus ihrer Belegung lässt sich nach Abschluss der Berechnung die Lösung des Problems rekonstruieren. Die Variablen, die die Eingabe des Modells repräsentieren, sind in Tabelle 8 beschrieben.

Variable	Bedingung	Beschreibung
X_{iljk}	$X_{iljk} = \begin{cases} 1, & k \text{ ist Nachfolger von } j. \\ 0, & \text{sonst.} \end{cases}$	Wird Auftrag j vor k auf Bearbeitungsstufe i auf Maschine l gefertigt?
C_{ij}	$C_{ij} \in \mathbb{N}_0$	Fertigstellungszeitpunkt von Auftrag j auf Stufe i.
C_{max}	$C_{max} \in \mathbb{N}_0$	Makespan.

Tabelle 7: Entscheidungsvariablen des ILP.

Variable	Bedingung	Beschreibung
n	$n > 0$	Anzahl der Aufträge.
m	$m > 0$	Anzahl der Stufen.
m_i	$m_i > 0 \; \forall \; i \in [0, m)$	Anzahl der Maschinen auf Stufe i.
N	$N = \{x \mid x \in \mathbb{N} \land x \leq n\}$	Menge aller Aufträge. Abweichend von der Definition in dieser Arbeit beginnen die Aufträge an Index 1, um Index 0 für benötigte Dummy-Aufträge freizuhalten.
M	$M = \{x \mid x \in \mathbb{N}_0 \land x < m\}$	Menge aller Stufen.
M_i	$M_{i'} = \{x \mid x \in \mathbb{N}_0 \land x < m_{i'}\} \; \forall \; i' \in M$	Menge aller Maschinen auf Stufe i.
p_{ilj}	$p_{ilj} \in \mathbb{N}_0 \; \forall \; i \in M, l \in M_i, j \in N$	Prozesszeit von Auftrag j auf Maschine l auf Bearbeitungsstufe i.
$skip_{ij}$	$skip_{ij} \in \mathbb{B} \; \forall \; i \in M, j \in N$	$skip_{ij} = \bot$ g.d.w. Auftrag j auf Bearbeitungsstufe i gefertigt werden muss. Andernfalls wird Stufe i übersprungen.
E_{ilj}	$E_{ilj} \in \mathbb{B} \; \forall \; i \in M, l \in M_i, j \in N$	Maschinenqualifikationen. $E_{ilj} = \top$ g.d.w. Auftrag j auf Maschine l auf Bearbeitungsstufe i gefertigt werden kann.
V	$V \in \mathbb{N}, V \gg \sum_{j \in N} p_{ilj} \; \forall i \; \in M, l \in M_i$	Große Zahl, um Nebenbedingungen bei fehlenden Voraussetzungen trivial wahr werden zu lassen.

Tabelle 8: Variablen des ILP.

4.4.2 Formulierung des ILP

Es wird nun die Formulierung der Zielfunktion und der Nebenbedingungen aufgezeigt und die einzelnen Ausdrücke anschließend erläutert.

Als Zielfunktion soll der Makespan minimiert werden, diese lautet:

$$min\ C_{max} \tag{11}$$

Die Nebenbedingungen lauten:

$$\sum_{\substack{j \in N \cup \{0\} \\ j \neq k, \neg skip_{ij}}} \sum_{\substack{l \in M_i \\ E_{ilj}, E_{ilk}}} X_{iljk} = 1, \quad i \in M, k \in N, \neg skip_{ik}, \tag{12}$$

$$\sum_{\substack{j \in N \\ j \neq k, \neg skip_{ij}}} \sum_{\substack{l \in M_i \\ E_{ilj}, E_{ilk}}} X_{ilkj} \leq 1, \quad i \in M, k \in N, \neg skip_{ik}, \tag{13}$$

$$\sum_{\substack{k \in N \\ \neg skip_{ik}, E_{ilk}}} X_{il0k} \leq 1, \quad i \in M, l \in M_i, \tag{14}$$

$$\sum_{\substack{h \in N \cup \{0\}, \\ h \neq k, h \neq j \\ \neg skip_{ih}, E_{ilh}}} X_{ilhj} \geq X_{iljk}, \quad i \in M, l \in M_i, j, k \in N, j \neq k, \neg skip_{ij}, \neg skip_{ik}, E_{ilj}, E_{ilk}, \tag{15}$$

$$\sum_{\substack{l \in M_i \\ E_{ilj}, E_{ilk}}} (X_{iljk} + X_{ilkj}) \leq 1, \quad i \in M, j \in N, k = j+1, \ldots, n, \neg skip_{ij}, \neg skip_{ik}, \tag{16}$$

$$C_{i0} = 0, \quad i \in M, \tag{17}$$

$$\begin{aligned} &C_{ik} - p_{ilk} \geq C_{ij} - V \cdot (1 - X_{iljk}), \\ &\quad i \in M, l \in M_i, j \in N \cup \{0\}, k \in N, j \neq k, \\ &\quad \neg skip_{ij}, \neg skip_{ik}, E_{ilj}, E_{ilk}, \end{aligned} \tag{18}$$

$$\begin{aligned} &C_{ik} - p_{ilk} \geq C_{i',k} - V \cdot (1 - X_{iljk}), \\ &\quad i \in M \setminus \{0\}, i' \in [0, i-1], l \in M_i, j \in N \cup \{0\}, k \in N, j \neq k, \\ &\quad \neg skip_{ij}, \neg skip_{ik}, \neg skip_{i'k}, E_{ilj}, E_{ilk}, \end{aligned} \tag{19}$$

$$C_{ij} \geq 0, \quad i \in M, j \in N, \neg skip_{ij}, \tag{20}$$

$$C_{max} \geq C_{ij}, \quad j \in N, i = \operatorname*{argmax}_{i \in M} \{\neg skip_{ij}\} \tag{21}$$

$$X_{iljk} \in \{0, 1\}, \quad i \in M, l \in M_i, j \in N \cup \{0\}, k \in N, j \neq k. \tag{22}$$

Die Entscheidungsvariablen X_{iljk} bestimmen die Auftragsketten auf den Maschinen. Die Nebenbedingung (22) beschränkt X_{iljk} auf den binären Wertebereich. $X_{iljk} = 1$ soll genau dann gelten, wenn Auftrag k direkter Nachfolger von Auftrag j auf Maschine l auf Bearbeitungsstufe i ist. Um auch für den ersten Auftrag auf einer Maschine diese Verknüpfung herstellen zu können, werden von Ruiz et al. [13] an Index $j = 0$ sogenannte *Dummy*-Aufträge eingeführt. Entsprechend werden in dieser Arbeit alle Aufträge um einen Index auf den Bereich $[1, n]$ verschoben, um den Index 0 für die Dummy-Aufträge freizuhalten. Es werden nur die tatsächlich möglichen X_{iljk} generiert. So wird der Vorgänger j aus der Vereinigungsmenge $j \in N \cup \{0\}$ aus Aufträgen und Dummy-Auftrag, der Nachfolger allerdings nur aus den Aufträgen $k \in N$ gewählt. Weiterhin gilt $j \neq k$, da ein Auftrag nicht auf sich selbst folgen kann. Weiterhin wird X_{iljk} nur dann betrachtet, wenn j und k auf Stufe i tatsächlich bearbeitet werden müssen und auf der Maschine l qualifiziert sind. Die Semantik der X_{iljk} wird in den Nebenbedingungen (12-16) definiert.

Zunächst wird in Nebenbedingung (12) beschränkt, dass jeder Auftrag $k \in N$ auf jeder Stufe $i \in M$ genau einen Vorgängerauftrag $j \in N \cup \{0\}$ besitzt. Dieser kann auch ein Dummy-Auftrag sein, wenn k der erste Auftrag auf der Maschine ist. Die Umkehrung, dass k maximal einen Nachfolger $j \in N$ pro Stufe besitzen kann, wird in Nebenbedingung (13) definiert. j kann hier kein Dummy-Auftrag sein, da dieser immer der erste auf der Maschine sein muss. Die Summe wird 1, wenn k einen Nachfolger j auf der Maschine l hat, oder 0, wenn k der letzte Auftrag auf der Maschine ist. Analog wird für den Dummy-Auftrag die Anzahl der Nachfolger pro Maschine in Nebenbedingung (14) auf 1 begrenzt. Nebenbedingung (15) sorgt dafür, dass Auftrage $j \in N$ auf Maschine $l \in M_i$ eingeplant sein muss, wenn Auftrag $k \in N$ auf ebendieser Maschine auf Auftrag j folgt. Dies sorgt dafür, dass die konstruierte Auftragskette aus den X_{iljk} alle auf derselben Maschine eingeplant werden. Die Nebenbedingung (16) verhindert direkte Zirkelbezüge in den Nachfolgerschaften. Auftrag j kann daher nicht gleichzeitig Nachfolger und Vorgänger von Auftrag k sein. k indiziert hier durch die Symmetrie nur die Aufträge, deren Index größer j ist. Die Nebenbedingung ist für die Zulässigkeit der Lösung nicht nötig, da die Linearität der Auftragsfolge durch die Bedingungen an die Endzeiten C_{ij} beschränkt ist und verhindert auch nicht die Bildung von Kreisen mit mehr als zwei Aufträgen. Allerdings schränkt sie die Anzahl der zu betrachtenden Lösungskandidaten stark ein und beschleunigt damit die Berechnung. Dies ist ein Beispiel dafür, dass das Hinzufügen von Nebenbedingungen auch zu einer Vereinfachung des Problems führen kann.

Die Entscheidungsvariablen C_{ij} bestimmen die Endzeiten der eingeplanten Aufträge. Aus der Kombination von X_{iljk} und C_{ij} lässt sich die konkrete Allokation eines Auftrages auf einer Maschine ableiten. Es gilt

$$\exists\ j \in N \cup \{0\} : X_{iljk} = 1 \quad \Longrightarrow \quad \text{Auftrag } k \text{ ist auf Maschine } l \text{ auf Bearbeitungsstufe } i \text{ in der Zeit von } C_{ij} - p_{ilj} \text{ bis } C_{ij} \text{ eingeplant.}$$

Die Nebenbedingungen (17-20) definieren die Semantik der C_{ij}. Nebenbedingung (20) beschränkt den Wertebereich der C_{ij} und sorgt dafür, dass kein Auftrag vor $t = 0$ beendet sein kann. Da $p_{ilj} \geq 0\ \forall i \in M, l \in M_i, j \in N$ gilt, liegt die Startzeit immer vor der Endzeit. Nebenbedingung (17) setzt die Endzeit aller Dummy-Aufträge auf 0. Dies markiert die Startzeit der Maschinen. Die Nebenbedingung (18) verhindert das Überlappen von Aufträgen auf derselben Maschine. Die Startzeit $C_{ik} - p_{ilk}$ des Auftrags $k \in N$ auf Maschine $l \in M_i$ auf Bearbeitungsstufe $i \in M$ darf nicht vor der Endzeit des Vorgängerauftrages $k \in N \cup \{0\}$, der auch ein Dummy-Auftrag sein kann, liegen. Der Term $(1 - X_{iljk})$ wird genau dann null, wenn k Nachfolger von j ist. Ist dies nicht der Fall, wird der Term 1 und die Multiplikation mit $-V$ macht den Ausdruck für alle Werte von C_{ik}, C_{ij} und p_{ilj} bei ausreichend großem V wahr. Nebenbedingung (19) definiert die Beziehung der Start- und Endzeiten desselben Auftrags beim Übergang zwischen zwei Stufen. Bei diesem darf die Startzeit $C_{ik} - p_{ilk}$ des Auftrags $k \in N$ auf der Stufe $i \in M \setminus 0$ auf Maschine $l \in M_i$ nicht kleiner sein als die Endzeit auf der vorherigen Stufe. Da Aufträge einzelne vorherige Stufen $i' < i$ überspringen können, werden die Nebenbedingungen nur dann angelegt, wenn $\neg skip_{i'k}$ gilt.

Nebenbedingung (21) definiert die in der Zielfunktion (11) zu minimierende Entscheidungsvariable C_{max}. Der Makespan ist definiert als

$$C_{max} = \max_{\substack{i \in M \\ j \in N}} \{C_{ij}\}\,.$$

Da die Maximums-Bedingung nicht direkt im ILP ausgedrückt werden kann, wird stattdessen die Nebenbedingung (21) definiert, dass $C_{max} \geq C_{ij}$ für alle Stufen $i \in M$ und Aufträge $j \in N$ gelten muss. Da C_{max} minimiert wird, ist die obere Schranke durch den größten Wert von C_{ij} scharf.

5 Evaluation

Der entwickelte Evolutionäre Algorithmus soll nun auf Testinstanzen evaluiert werden. Dieser kann hybride Flow Shops mit maschinenspezifischen Prozesszeiten und beliebig vielen Bearbeitungsstufen lösen. Da die genetischen Operatoren aber nur auf die erste Stufe angewendet werden und alle Weiteren mittels der Anstellstrategie ECT (Alg. 18) eingeplant werden, wird ein hybrider Flow Shop mit zwei Bearbeitungsstufen betrachtet. Der Algorithmus setzt des Weiteren die Nebenbedingung *skip* und M_j um.

5.1 Benchmarks

Aus den existierenden Benchmarks aus Tabelle 3, die im Abschnitt 3 besprochen wurden, wird der Datensatz von Ruiz et al. [13] verwendet. Dieser enthält alle nötigen Daten, das heißt maschinenspezifische Prozesszeiten und Qualifikationen, sowie Daten zum Überspringen von Stufen. Zunächst werden aus dem gewählten Benchmark Testinstanzen ausgewählt und mittels des entwickelten ILP gelöst, um die Ergebnisse der verschiedenen Konfigurationen des Evolutionären Algorithmus vergleichen zu können. Der Benchmark von Ruiz et al. [13] enthält Datensätze mit verschiedenen Ausprägungen der Nebenbedingungen. Tabelle 9 zeigt die Konfigurationen der Testinstanzen im Benchmark. Die grau hinterlegten Einträge sind für diese Arbeit irrelevant, da sie Daten für Nebenbedingungen definieren, die bislang nicht im Algorithmus berücksichtigt werden. Diese finden entsprechend in der Auswertung keine Anwendung. Die unterstrichenen Werte zeigen die genutzte Konfiguration:

Faktor	**Werte**
Anzahl Aufträge	**50**, 100
Anzahl Bearbeitungsstufen	**4**, 8
Anzahl Maschinen pro Stufe	2, **4**
skip-Wahrscheinlichkeit pro Auftrag	0%, **50%**
Maschinenqualifikationswahrscheinlichkeit	**50%**, 100%
Verteilung der Maschinenstartzeiten	0, U[1,200]
Verteilung der Umrüstzeiten	U[25, 74], U[75, 125]
Wahrscheinlichkeit für antizipatorische Umrüstzeiten	U[0, 50]%, U[50, 100]%
Verteilung der lag-Zeiten	U[1, 99], U[-99, 99]
Anzahl Vorgängeraufträge	0, U[1, 3]

Tabelle 9: Konfigurationen der Benchmarks nach Ruiz et al. [13].

Für jede Konfiguration existieren drei Datensätze, sodass bei Vernachlässigung von fünf Faktoren mit je zwei Ausprägungen $3 \times 2^5 = 96$ mögliche Testinstanzen verbleiben. Aus diesen wurden zehn Instanzen mit dem entwickelten ILP gelöst. Da der Algorithmus nur auf der ersten Bearbeitungsstufe genetische Operationen durchführt und für die fol-

genden Stufen lediglich ECT (Alg. 18) genutzt wird, wurden nur die ersten beiden Stufen eingeplant. Die Anzahl der Aufträge wurde auf 30 reduziert. Es ergibt sich die folgende Problemformulierung:

$$FH2, (R4^1, R4^2) \mid skip, M_j \mid C_{max}$$

Da die Daten von denen aus Ruiz et al. [13] durch die nicht verwendeten Nebenbedingungen und vernachlässigten Stufen und Aufträge abweicht, können dessen Ergebnisse der exakten Optimierung nicht verwendet werden. Die für alle Optimierungen genutzte Hardware wird in Tabelle 10 beschrieben.

Prozessor	**RAM**	**OS**	**Software**
Intel®Xeon®W-2265 CPU @ 3,50GHz, 24 Threads	64GB	Windows 10 Pro 22H2	Python 3.11, Gurobi 10.0.1

Tabelle 10: Genutzte Hardware für die Optimierungsläufe des ILP und der Parameteroptimierung.

Die Ergebnisse der exakten Optimierung sind in Tabelle 11 dargestellt. Das Modell wurde mit Gurobi 10 gelöst. V wurde als 4.000 gewählt. Die Laufzeiten zum Finden der optimalen Lösung schwanken dabei zwischen wenigen Sekunden und einigen Stunden.

Instanz	**C_{max}**	**Laufzeit**	**Instanz**	**C_{max}**	**Laufzeit**
0	171	13h	5	201	13h
1	238	12h	6	155	26s
2	250	1h	7	198	4h
3	217	1h	8	145	10m
4	205	2m	9	190	1h

Tabelle 11: Ergebnisse der exakten Optimierung.

5.2 Vergleich der Lösungen

Mit den Optima kann nun für die folgenden Auswertungen der heuristischen Optimierung ein Maß definiert werden, das die Abweichung der Lösung prozentual zum Optimum angibt. Dies ermöglicht den Vergleich der Läufe auf unterschiedlichen Instanzen. Der relative Abstand (*Relative Percentage Deviation - RPD*) berechnet sie wie folgt, wobei *heu* die heuristische Lösung und *opt* das Optimum darstellt [60, S. 6]:

$$RPD = \frac{heu - opt}{opt} \geq 0$$

Die Grafiken der im Folgenden beschriebenen Auswertungen mittels der Gittersuche und der modellbasierten Parameteroptimierung zeigen entsprechend ausschließlich relative Abweichungen zum Optimum. Als Abbruchkriterium für den Evolutionären Algorithmus

wurde die Anzahl der Zielfunktionsauswertungen auf 1.000 gesetzt. Das Auswerten der Zielfunktion ist der teuerste Schritt in der Optimierung und ist einem Abbruchkriterium über die Laufzeit des Algorithmus vorzuziehen, da die Ergebnisse reproduzierbar auf anderer Hardware werden und unabhängig von anderen, parallel laufenden Prozessen sind.

Zum Vergleich verschiedener Parameterkonfigurationen werden gepaarte t-Tests verwendet. Diese eignen sich, um zwei normalverteilte Stichproben X, Y miteinander zu vergleichen und über einen signifikanten Unterschied zu entscheiden [89, S. 379 ff.]. Hierbei werden die Testkonfigurationen gepaart betrachtet, indem immer gleiche Konfigurationen, die sich nur im untersuchten Parameter unterscheiden, voneinander subtrahiert werden [90, S. 606]. Daraus ergibt sich eine Normalverteilung $D = X - Y$. Der gepaarte t-Test entscheidet anschließend, ob die Differenzenverteilung im Mittel signifikant von null verschieden ist, das heißt, dass die eine Stichprobe signifikant bessere oder schlechtere Ergebnisse liefert [91, S. 1]. Die Nullhypothese H_0 und die Alternativhypothese H_1 lauten entsprechend, dass die Stichproben identisch bzw. signifikant verschieden sind [92, S. 17]:

$$H_0 : \mu_D = 0 \qquad H_1 : \mu_D \neq 0$$

Für die Anwendung des t-Tests müssen die Stichproben statistisch unabhängig sein, was durch die unabhängige Durchführung der Experimente gegeben ist. Der Weiteren muss die Stichprobe der Differenzen annähernd gleichverteilt sein [92, S. 17], was auf alle folgenden Auswertungen zutrifft.

5.3 Gittersuche

Zunächst wurde eine Gittersuche auf dem Parameterraum durchgeführt, um eine Einordnung der Beziehungen zwischen den Parametern und ihrem Einfluss auf das Optimierungsergebnis zu erhalten. Dazu müssen Werte für die numerischen Parameter gewählt werden. Wie bereits in Abschnitt 2.3.1 erläutert, ist für eine Analyse der Beziehungen zwischen den Parametern eine äquidistante Wahl vorzuziehen. Für alle Parameter sollen gleich viele Werte eingesetzt werden. Die Anzahl der durchzuführenden Optimierungsläufe N berechnet sich mit dem Parameter n als Anzahl zu wählender Werte für die vier numerischen Parameter unter Berücksichtigung der jeweils zwei möglichen Varianten für die fünf Operatoren als

$$N = 2^5 \times n^4.$$

Der Parameter n geht dabei quartisch in die Anzahl der durchzuführenden Optimierungsläufe ein, sodass viele Werte die Anzahl rasant ansteigen lässt. Es wurde daher $n = 3$ gewählt, wodurch $N = 2.592$ Optimierungsläufe durchzuführen sind. Die gewählten Parameter sind in Tabelle 12 dargestellt.

Parameter	Werte
SolutionGenerator	SolutionGeneratorRandomMachines, SolutionGeneratorBalancedMachines
MutationOperator	MutationOperatorShift, MutationOperatorSwap
CrossoverOperator	CrossoverOperatorOrderBased, CrossoverOperatorMachineAllocation
ParentsSelectionOperator	SelectionOperatorRandom, SelectionOperatorTournament
SurvivalSelectionOperator	SelectionOperatorTournament, SelectionOperatorTruncation
pop_size	{ 1, 50, 100 }
par_size	{ 0.1, 0.55, 1.0 }
p_c	{ 0.0, 0.50, 1.0 }
p_m	{ 0.0, 0.50, 1.0 }

Tabelle 12: Gewählte Parameter für die Gittersuche.

Jeder Optimierungslauf besteht aus der Auswertung aller zehn Testinstanzen, wobei jede Auswertung wegen der im Algorithmus vorherrschenden Stochastik 20-mal repliziert und gemittelt wurde. Die Replikationen wurden dabei parallel auf den zur Verfügung stehenden CPU-Threads durchgeführt. Es ergeben sich insgesamt 518.400 Läufe. Jeder Optimierungslauf gibt anschließend die zehn Ergebnisse relativ zu den Optima zurück. Die zehn besten Konfigurationen sind in Tabelle 13 zu sehen. Die mittlere RPD über die zehn Testinstanzen ist als Qualitätsmaß angegeben und bestimmt das Ranking der Konfigurationen. Im Folgenden wird nun der Einfluss der einzelnen Operatoren und insbesondere auch Wechselwirkungen zwischen den Parametern betrachtet.

Solution Generator	Mutation Operator	Crossover Operator	Survival Selection	Parents Selection	pop_ size	par_ size	p_c	p_m	mean RPD
Balanced	Shift	MA	Truncation	Tournament	100	0.55	0.0	1.0	0.0897
Balanced	Shift	MA	Truncation	Tournament	100	1.0	0.5	1.0	0.0900
Balanced	Shift	MA	Tournament	Random	100	0.55	0.0	1.0	0.0931
Balanced	Shift	MA	Tournament	Random	100	0.55	0.5	1.0	0.0941
Balanced	Shift	MA	Truncation	Tournament	100	0.55	0.0	0.5	0.0948
Balanced	Shift	MA	Tournament	Random	100	0.55	0.0	0.5	0.0949
Balanced	Shift	OBX	Truncation	Tournament	100	0.55	0.0	1.0	0.0951
Balanced	Shift	MA	Truncation	Tournament	50	0.55	0.5	1.0	0.0952
Balanced	Shift	MA	Tournament	Random	100	1.0	0.0	0.5	0.0952
Balanced	Shift	OBX	Tournament	Tournament	100	0.55	0.0	0.5	0.0956

Tabelle 13: Ergebnisse der Gittersuche.

5.3.1 Lösungsgeneratoren

Zunächst wird der Einfluss der Initiallösungen auf die Güte der resultierenden Ergebnisse betrachtet. Zur Verfügung stehen die Operatoren *SolutionGeneratorRandomMachines (Random)* (Alg. 9) und *SolutionGeneratorBalancedMachines (Balanced)* (Alg. 10). Wie in Abschnitt 4.2.1 erläutert, bildet der zufällige Operator den gesamten Lösungsraum ab und generiert eine diversere Population, während der ausbalancierte tendenziell bessere Initiallösungen in Bezug auf den Zielfunktionswert liefert. Abbildung 18a zeigt die Zielfunktionswerte des jeweils besten Individuums aus der Population nach der Erzeugung durch die Generatoren vor der Optimierung. Hier ist klar zu sehen, dass der balancierte Generator deutlich bessere Individuen in Bezug auf den Zielfunktionswert generiert. Abbildung 18c zeigt die Differenz der beiden Kurven. *Balanced* dominiert *Random* nach dem gepaarten t-Test signifikant mit $\alpha = 0.05$. Für alle weiteren Auswertungen wurden die Läufe mit $p_m = 0.0 \wedge p_c = 0.0$ aus der Betrachtung ausgenommen, da sie keine Operationen auf der Population durchführen. Das Signifikanzniveau wird durchgehend als $\alpha = 0.05$ gewählt.

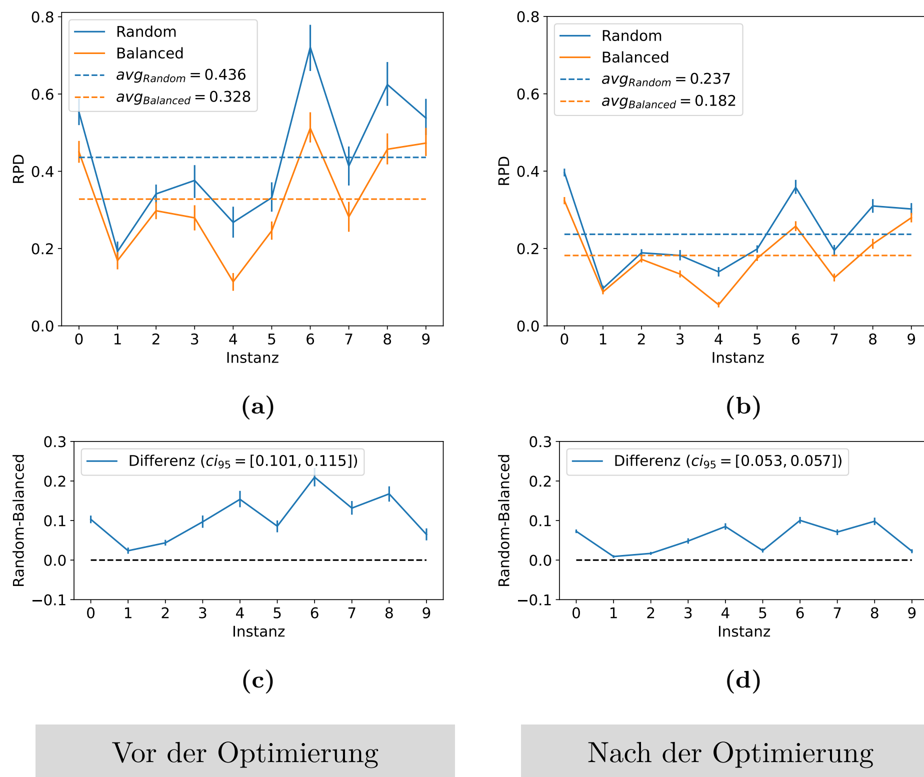


Abbildung 18: Vergleich der Lösungsgeneratoren vor und nach der Optimierung.

Betrachtet man nun den Unterschied zwischen den Lösungsgeneratoren nach der Optimierung in Abb. 18b, so ist dieser geringer, aber weiterhin signifikant. Insbesondere ist in Abb. 18d zu sehen, dass die Konfidenzintervalle der Differenz deutlich kleiner ausfallen, was den Unterschied zwischen den beiden Verfahren weiter betont.

5.3.2 Selektionsoperatoren

Bei den Selektionsoperatoren wird zwischen der Eltern- und der Überlebensselektion unterschieden. Die Elternselektion wählt Individuen aus, die anschließend durch Crossover und/oder Mutation verändert werden. Die Überlebensselektion wählt Individuen am Ende einer Schleifeniteration aus, die in die Nächste übernommen werden. In Abbildung 19 ist zu sehen, dass in beiden Fällen die Turnierselektion gegenüber ihrer jeweiligen Alternative unterliegt. Obwohl jeweils der Unterschied zwischen den jeweiligen Verfahren signifikant ist, ist ihre mittlere Abweichung dennoch sehr gering. In Abb. 19c ist beispielsweise zu sehen, dass die Turnierselektion für Instanz 6 besser abschneidet als die zufällige Wahl.

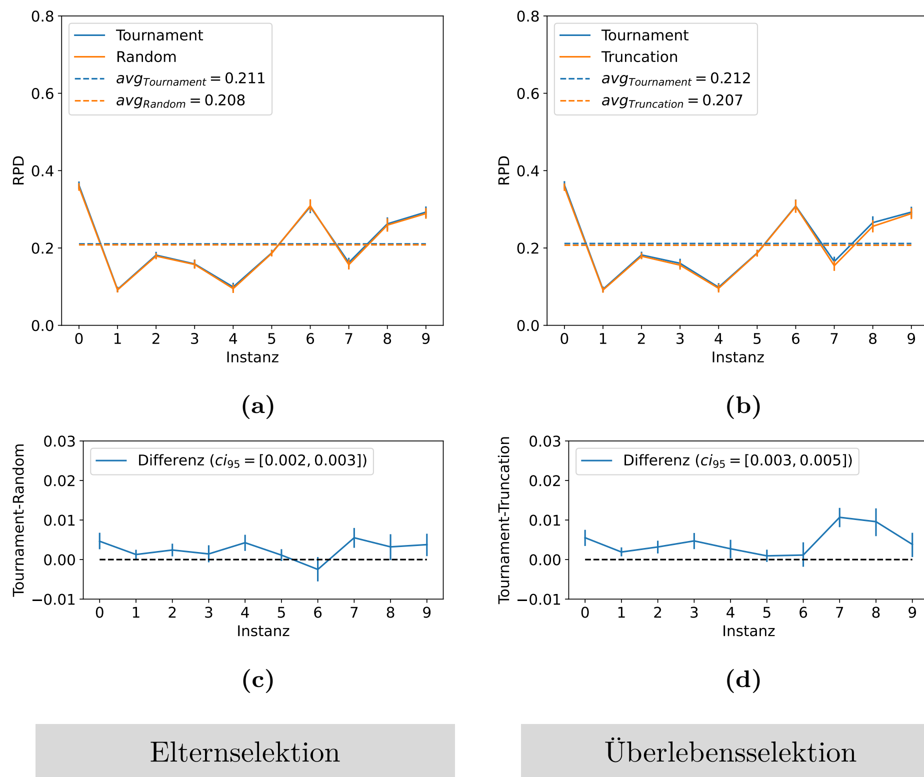


Abbildung 19: Vergleich der Selektionsoperatoren.

Zu beachten ist hier die geänderte y-Achsenskalierung in den Abbildungen 19c und 19d um Faktor 10, um dem sehr geringen Unterschied der beiden Verfahren Rechnung zu tragen.

5.3.3 Genetische Operatoren

Bei den genetischen Operatoren werden Crossover und Mutation in Abbildung 20 zunächst isoliert voneinander betrachtet. Bei den Mutationsoperatoren dominiert *Shift* klar *Swap*. Auch ohne Anwendung von Crossoveroperatoren konvergiert der Algorithmus gut gegen das Optimum. Bei den Crossoveroperatoren dominiert der *CrossoverOperator-OrderBased (OBX)* (Alg. 14) den Operator *CrossoverOperatorMachineAllocation (MA)* (Alg. 15).

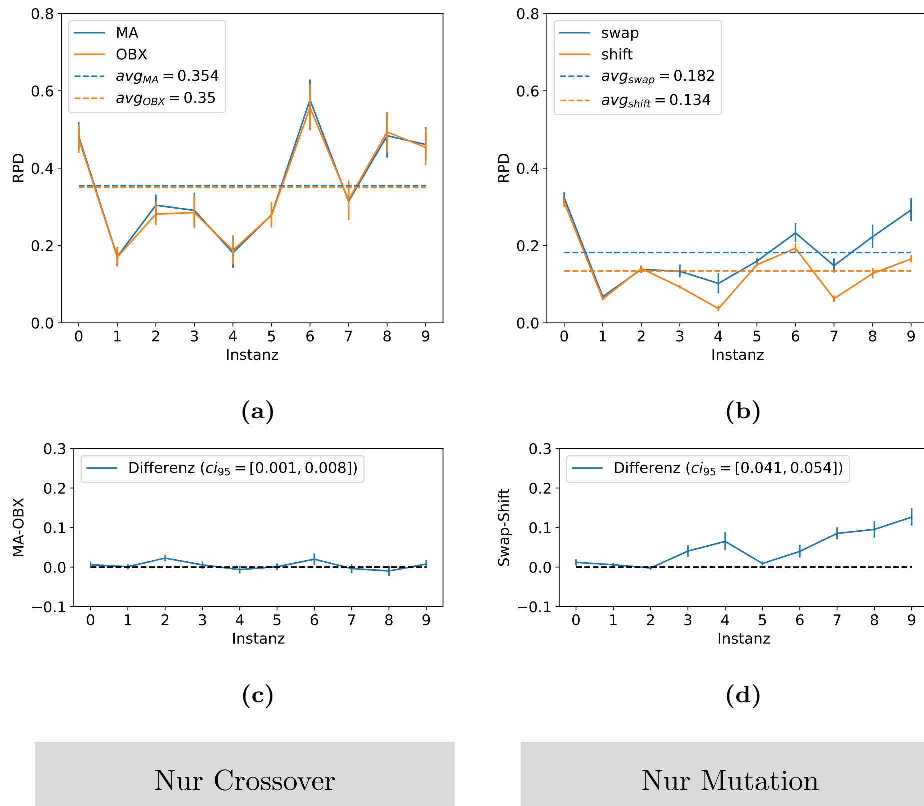


Abbildung 20: Isolierter Vergleich der Crossover- und Mutationsoperatoren.

Es ist zu beachten, dass alle drei Konfigurationen unter den besten zehn, die eine Crossoverwahrscheinlichkeit $p_c > 0$ gewählt haben, dennoch den Operator *MA* nutzen. Die Differenz ist signifikant nach gepaartem t-Test, fällt aber ähnlich zu den Selektionsoperatoren klein aus. Grund dafür ist die schlechte Konvergenz des Algorithmus ohne Mutation, die in Abbildung 21 deutlich wird.

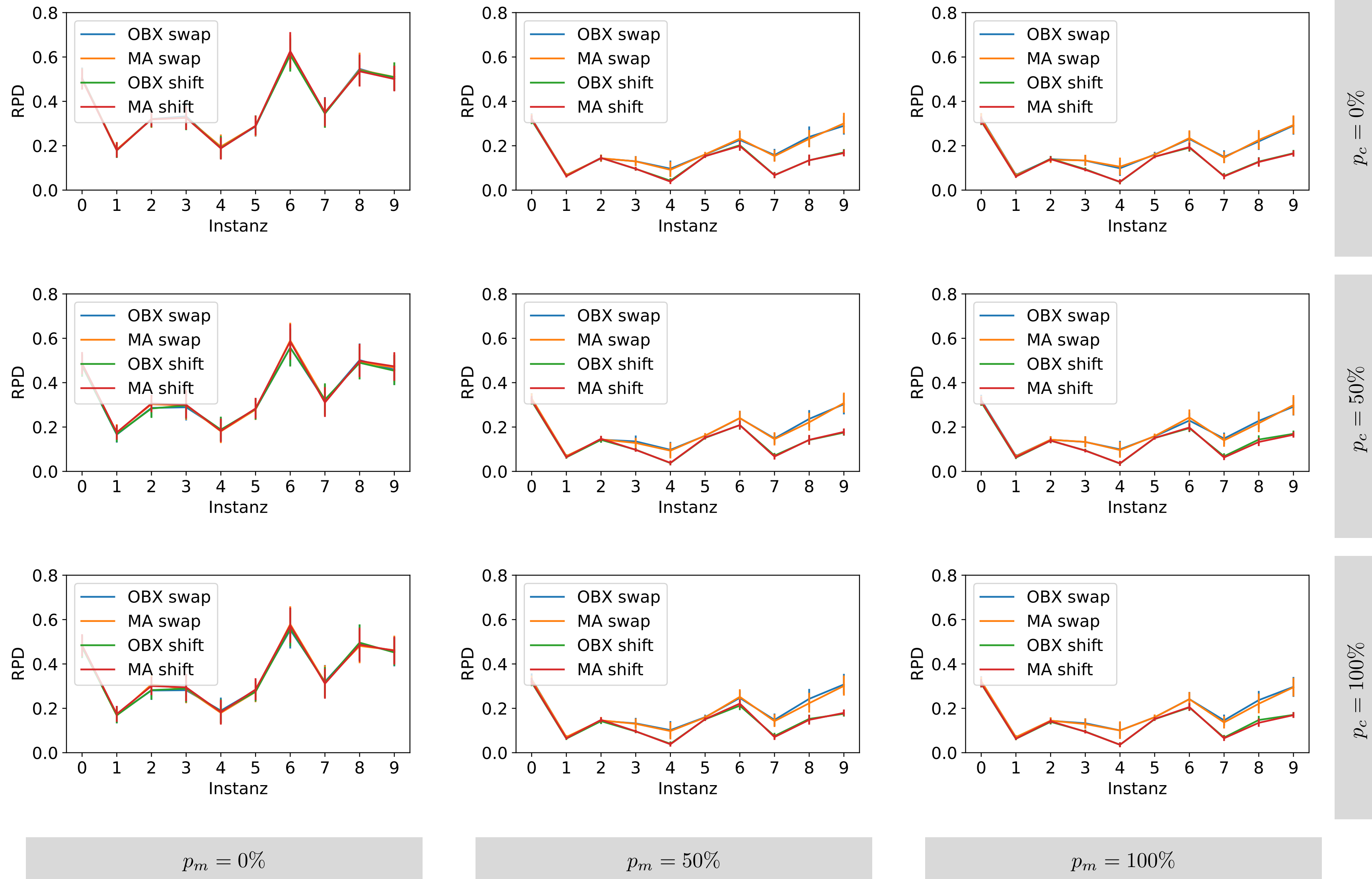


Abbildung 21: GG-Plot der Beziehungen zwischen Crossover- und Mutationsoperatoren und -wahrscheinlichkeiten.

Gezeigt wird hier das Zusammenspiel von Crossover und Mutation bei den verschiedenen Wahrscheinlichkeitsniveaus. Über alle Wahrscheinlichkeitsniveaus der Mutation gemittelt konnte ein signifikanter Unterschied zwischen $p_c = 0\%$ und $p_c = 50\%$, sowie $p_c = 0\%$ und $p_c = 100\%$ gezeigt werden, nicht jedoch zwischen $p_c = 50\%$ und $p_c = 100\%$. Hierbei ist jeweils eine niedrigere Crossoverwahrscheinlichkeit präferabel. Zwischen den Wahrscheinlichkeitsniveaus der Mutation p_m konnte gemittelt über alle Wahrscheinlichkeitsniveaus des Crossovers paarweise signifikant gezeigt werden, dass eine höhere Mutationswahrscheinlichkeit vorzuziehen ist. Die Ergebnisse sind in Abbildung 22 getrennt nach Crossover- und Mutationswahrscheinlichkeiten zusammengefasst.

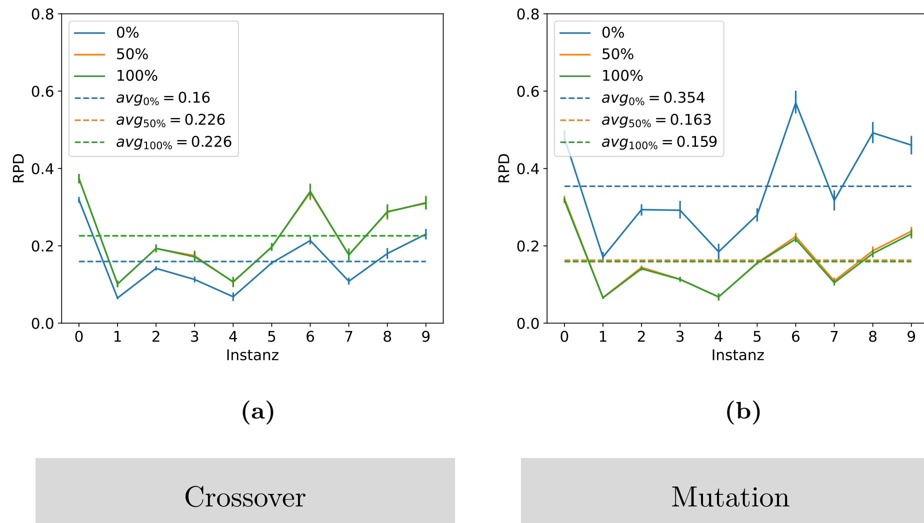


Abbildung 22: Vergleich der Crossover- und Mutationswahrscheinlichkeiten.

Hier ist nochmal der geringe Unterschied zwischen $p_c = 50\%$ und $p_c = 100\%$, sowie zwischen $p_m = 50\%$ und $p_m = 100\%$ zu unterstreichen. $p_c = 0\%$ und $p_m = 0\%$ stechen hingegen stark heraus.

5.3.4 Populationsgröße und Elternschaft

Bei der Populationsgröße in Abbildung 23a konnten erneut signifikante Unterschiede zwischen den drei Wahrscheinlichkeitsniveaus gefunden werden. Eine größere Population führt dabei paarweise zu besseren Ergebnissen. Insbesondere schneidet die Konfiguration mit nur einem Individuum in der Population besonders schlecht ab. Bei der Größe der Elternschaft in Abbildung 23b zeigen sich nur geringe Unterschiede. Zwischen 100% und 55% konnte kein signifikanter Unterschied gezeigt werden, 10% schneidet aber gegen beide anderen signifikant schlechter ab.

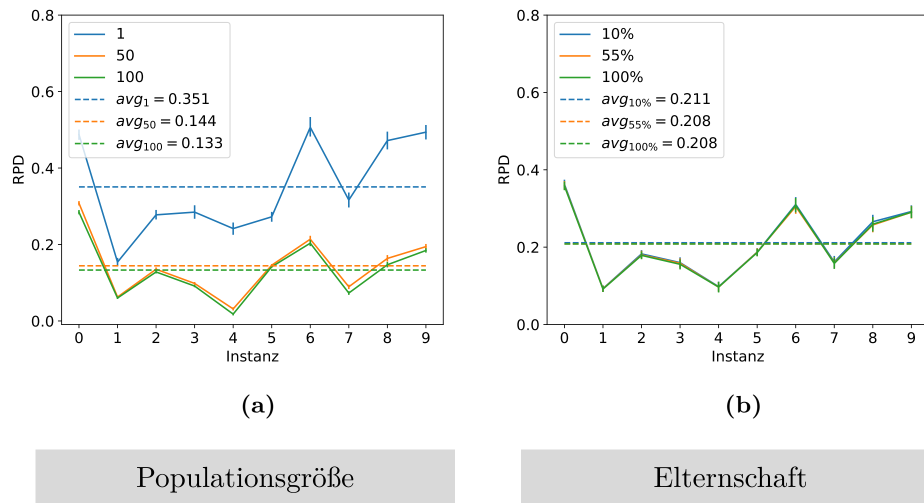


Abbildung 23: Isolierter Vergleich der Populationsgröße und Elternschaft.

Die Kombination dieser beiden Werte zeigt, dass eine breite Suche mit großer Population und großer Elternschaft einer Tiefen vorzuziehen ist. Das zeigt auch, dass der Evolutionäre Algorithmus aus seiner Anfangspopulation auf dem aktuellen Datensatz schnell, das heißt mit wenigen Schleifeniterationen, gegen das Optimum konvergiert. Ein GG-Plot der Zusammenhänge zwischen Populationsgröße und Größe der Elternschaft befindet sich im Anhang in Abbildung 28.

5.4 Modellbasierte Parameteroptimierung

Die modellbasierte Optimierung verspricht eine effizientere Suche durch Minimierung der auszuwertenden Parameterkonfigurationen. Die Kosten zum Fitten des Modells sind im Vergleich zur Laufzeit einer Parameterevaluation zu vernachlässigen, sodass ein Vorteil durch die Nutzung eines Surrogatmodells wahrscheinlich ist, insofern die Prädiktionen neuer Parameter auf dem Modell tatsächlich zu einer Konvergenz zum Optimum führt. Die modellbasierte Parameteroptimierung soll hier nicht neutral mit der Gittersuche verglichen werden, sondern dessen Ergebnisse nachoptimieren. Für die initiale Modellkonstruktion ist eine initiale Testmenge nötig, die noch ohne Modell erzeugt wird. Hierfür können die Ergebnisse der Gittersuche verwendet werden, da sie den Suchraum durch die äquidistante Wahl der Versuchspunkte gleichmäßig abbildet. Des Weiteren kann die Auswahl der verfügbaren Kombinatoren durch die Vorergebnisse eingeschränkt werden. Der Lösungsgenerator für zufällige Lösungen und der Mutationsoperator *Swap* wurden per *Blacklist* aus dem Repository entfernt, da sie klar durch die jeweilige Alternative dominiert wurden. Bei den verbleibenden Kombinatoren konnten in der Gittersuche auch signifikante Unterschiede festgestellt werden, jedoch waren diese nicht so eindeutig wie bei den zuvor genannten zwei. Die Wertebereiche für die numerischen Parameter wurden belassen. Da diese auf einer kontinuierlichen Skala liegen, wird das Surrogatmodell hier durch eine Regression die Zusammenhänge fitten. Die Vorgehensweise für die gesamte Parameteroptimierung inklusive der vorangegangenen Gittersuche ist in Abbildung 24 illustriert.

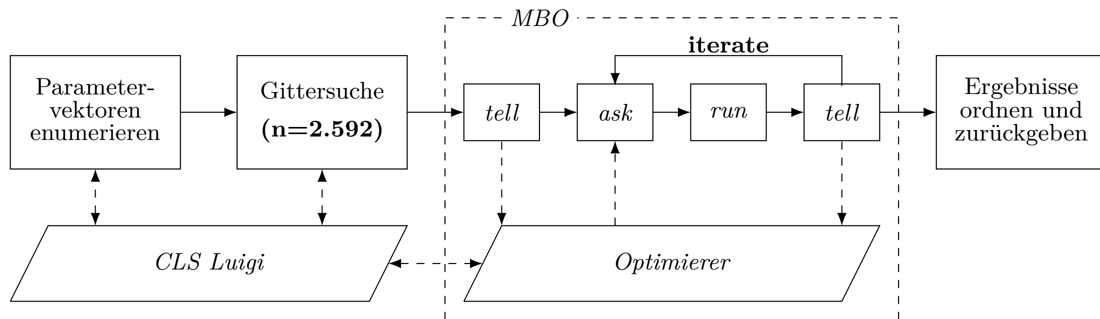


Abbildung 24: Ablauf der gesamten Parameteroptimierung.

Für die modellbasierte Optimierung wurde das in Abschnitt 2.3.2 besprochenen Framework *skopt* genutzt. Als Aquisitionsfunktion wurde die Funktion *gp_hedge* verwendet, die die Funktionen *Probability of Improvement (PI)*, *Expected Improvement (EI)* und *Lower Confidence Bound (LCB)* probabilistisch auswählt. Die Wahrscheinlichkeitsverteilung und der Parameter κ für *LCB* werden während der Optimierung zusätzlich optimiert. Die Konfiguration des Optimierers ist in Tabelle 14 dargestellt.

Parameter	Wert	Beschreibung
dimensions	<obj>	Wertevektoren für die Parameter des Evolutionären Algorithmus. Durch CLS ermittelt.
base_estimator	GP	Wählt den Gaussprozess als Regressor für das Fitting des Surrogatmodells.
n_initial_points	0	Führe kein initiales Sampling durch, da die Ergebnisse der Gittersuche als initiales Design vorgegeben werden.
acq_func	gp_hedge	Wähle aus den drei Aquisitionsfunktionen *PI*, *EI* und *LCB* in jeder Iteration probabilistisch ein Verfahren aus.
acq_optimizer	auto	Optimiere automatisch die Wahrscheinlichkeitsverteilung für die Aquisitionsfunktionen und den Parameter κ für *LCB*.
n_jobs	1	Führe immer nur eine Optimierung gleichzeitig durch. Die Replikationen des zu optimierenden Algorithmus werden weiter parallelisiert.
model_queue_size	None	Keine Beschränkung der Modellgröße. Das Fitting wird auf allen bisher getesteten Parametrisierungen durchgeführt.

Tabelle 14: Konfiguration des Parameteroptimierers.

Die Optimierungsschleife wurde an Stelle der eingebauten Funktion *gp_minimize* selbst implementiert, um mehr Kontrolle über die Optimierung zu erhalten. Im ersten Schritt werden dem Optimierer die Ergebnisse der Gittersuche über die Funktion *tell* übergeben. Nach dem ersten Fitting des Surrogatmodells erfolgt die eigentliche Optimierungsschleife, die die Phasen $ask \rightarrow run \rightarrow tell$ durchläuft. *ask* führt dabei die Prädiktion des nächsten Versuchspunktes unter Verwendung der Aquisitionsfunktion durch. *run* bezeichnet hier die Ausführung der Parameterkonfiguration, die selbst implementiert wurde. *tell* übergibt schließlich den neuen Funktionswert und führt ein re-Fitting des Modells durch. Nach Abschluss der Optimierung werden die Ergebnisse nach Zielfunktionswert geordnet und zurückgegeben. Als Terminierungskriterium wurde eine Zeitschranke von zehn Tagen gewählt, in denen **1.696** Iterationen durchgeführt wurden. Tabelle 15 zeigt die Ergebnisse der modellbasierten Optimierung. Lösungsgenerator und Mutationsoperator wurden wie beschrieben auf Grund der Ergebnisse der Gittersuche eingeschränkt. Alle gezeigten Läufe wurden mit dem Lösungsgenerator *SolutionGeneratorBalancedMachines* und dem Mutationsoperator *MutatationOperatorShift* durchgeführt. Der Crossoveroperator ist zwar in der Tabelle dargestellt, findet aber keine Anwendung, da die Crossoverwahrscheinlichkeiten p_c aller zehn Läufe null sind.

Iteration	Crossover Operator	Survival Selection	Parents Selection	pop_size	par_size	p_c	p_m	mean RPD
835	OBX	Truncation	Tournament	100	0.74	0.00	0.43	0.0878
959	OBX	Truncation	Tournament	100	0.72	0.00	0.43	0.0881
822	OBX	Truncation	Tournament	100	0.75	0.00	0.43	0.0892
1039	OBX	Truncation	Tournament	100	0.81	0.00	0.57	0.0894
746	OBX	Truncation	Tournament	100	0.74	0.00	0.42	0.0895
1542	OBX	Tournament	Random	100	0.40	0.00	0.79	0.0900
1078	OBX	Truncation	Tournament	100	0.76	0.00	0.46	0.0901
754	OBX	Truncation	Tournament	100	0.75	0.00	0.41	0.0901
1040	OBX	Truncation	Tournament	100	0.76	0.00	0.62	0.0903
1345	OBX	Tournament	Random	100	0.41	0.00	1.00	0.0903

Tabelle 15: Ergebnisse der modellbasierten Optimierung.

Die Ergebnisse bestätigen die Erkenntnis aus der Gittersuche, dass der Operator *SelectionOperatorTruncation* für die Überlebensselektion gegenüber der Turnierselektion vorzuziehen ist. Der Unterschied war gering, jedoch dominiert er die Turnierselektion wie in Abbildung 19b zu sehen für die Instanzen sieben und acht, was sich hier ebenso widerspiegelt. Bei der Elternselektion führt hier, entgegen der Erkenntnisse aus der Gittersuche, die Turnierselektion. Der Unterschied zwischen den beiden Optionen Turnierselektion und zufälliger Wahl war aber bereits bei der Gittersuche gering. Bei den numerischen Parametern bestätigen sich die Ergebnisse aus Abbildung 23 bezüglich der Populationsgröße und Elternschaft. Die Populationsgröße wurde im gegebenen Wertebereich maximiert, was nahelegt, dass eine noch größere Population möglicherweise noch bessere Ergebnisse liefern könnte. Bei der Größe der Elternschaft konnten bei der Gittersuche keine großen Unterschiede festgestellt werden. Analog sind unter den zehn besten Konfigurationen Werte im Bereich 40% und 75% zu finden. Die Crossoverwahrscheinlichkeit wurde in allen Läufen auf null gesetzt, was die Ergebnisse aus der Gittersuche stützt. Bei der Mutationswahrscheinlichkeit finden sich Werte zwischen 40% und 100%. Dazu sei gesagt, dass bei fehlendem Crossover eine geringere Mutationswahrscheinlichkeit äquivalent zu einer Reduktion der Elternschaftsgröße ist. Die im Mittel pro Iteration mutierten Individuen ergeben sich bei $p_c = 0$ als $par_size \times p_m$. Für die zehn besten Konfiguration sind dies 36 Individuen.

Die Konvergenz der Suche ist in Abbildung 25 zu sehen. Man beachte hier die auf Grund der Ausreißer zu Beginn der Suche gewählte logarithmische Skalierung der y-Achse. Bereits nach wenigen Iterationen werden von der modellbasierten Optimierung keine Konfigurationen mit hohem RPD mehr ausgewählt, weshalb die Ausreißer hauptsächlich am Anfang der Suche zu finden sind. Die Konvergenz der besten gefundenen Konfiguration über den Verlauf der Optimierung ist in Abb. 25 eingezeichnet. Die letzte Verbesserung wurde nach **835** Iterationen mit einem mittleren RPD von **0.0878** erzielt. Der Datenpunkt entspricht der ersten und besten Konfiguration in Tabelle 15. Die Konvergenz der RPD der getesteten Konfigurationen ist ebenso in Abb. 25 als Regressionskurve gezeigt. Es ist zu sehen, dass nach der letzten Verbesserung in der zweiten Hälfte der Suche die Punktwolke stagniert, sodass mit keiner konvergenzbedingten Verbesserung mehr zu rechnen ist. Durch

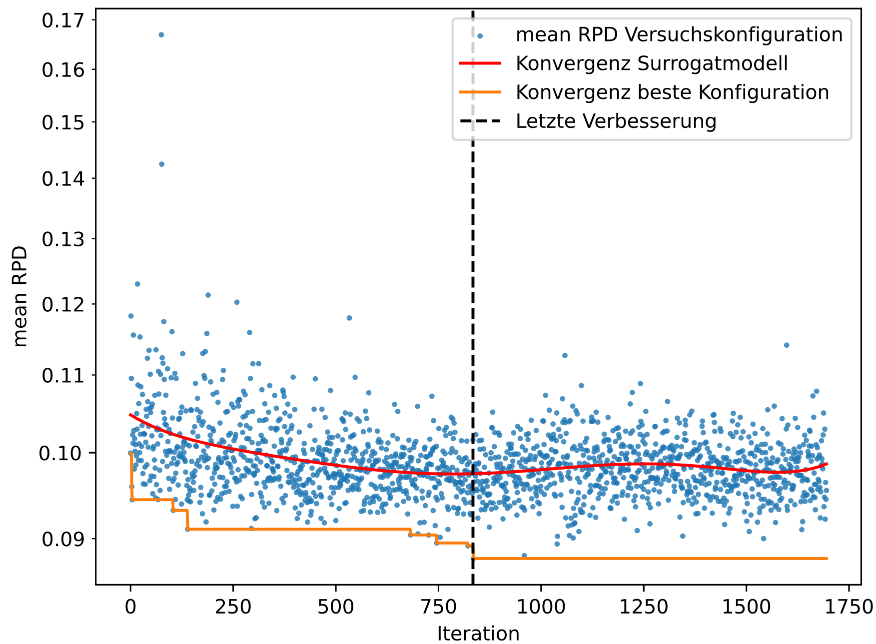


Abbildung 25: Konvergenzgraph der modellbasierten Optimierung.

die Verwendung der gemischten Aquisitionsfunktion werden auch nach Auffinden guter Lösungen weiter Bereiche des Lösungsraumes exploriert und Punkte zur Verbesserung der Modellqualität überprüft. Dies erklärt die Schwankung des RPD der getesteten Werte bis hin zur Verschlechterung im Mittel zum Ende der Suche.

Es sei noch einmal hervorgehoben, dass ein optimaler Zielfunktionswert durch die eingeschränkte Erreichbarkeit im Lösungsraum ggf. nicht möglich ist, sodass ein RPD 0.0878 möglicherweise bereits optimal ist. Da die heuristische Optimierung im Gegensatz zur exakten Optimierung aus Abschnitt 5.1 keine untere Schranke angibt, kann ein solcher Schluss aber nicht mit Sicherheit gezogen werden. Der Mutationsoperator *Shift* (Alg. 16) spannt zwar auf der ersten Bearbeitungsstufe, wie in Abschnitt 4.2.4 beschrieben, den gesamten Lösungsraum auf, jedoch führt die Verwendung der Stufenstrategie ECT (Alg. 18) zu einer eingeschränkten Erreichbarkeit auf der zweiten Bearbeitungsstufe, was die Abweichung vom Optimum erklärt.

6 Diskussion der Ergebnisse

Die Ergebnisse der beiden Parameteroptimierungen werden im Folgenden verglichen und im Kontext der Erkenntnisse aus der Literaturrecherche diskutiert. Alle Ergebnisse beziehen sich dabei explizit nur auf die vorliegenden Testdaten und eine Übertragbarkeit auf andere Instanzen und Problemklassen ist, wie im Literaturüberblick herausgestellt, nur eingeschränkt möglich, da die zu wählenden Parameter stark von den Charakteristika des Problems abhängen können.

Die Gittersuche mit äquidistanter Wahl der numerischen Parameter ermöglichte eine von den anderen Parametern isolierte Betrachtung einzelner Einflussfaktoren. Hier konnte der klare Vorteil von besseren Startlösungen für die gewählte Probleminstanz festgestellt werden. Bei den Mutationsoperatoren stellte sich der Operator *Shift* als überlegen heraus, obwohl die Stärke der Veränderung einzelner Individuen kleiner ist als bei *Swap*. Dies lässt sich durch den größeren Suchraum begründen, der die Erreichbarkeit besserer Lösungen gewährt.

Die modellbasierte Parameteroptimierung bestätigte im Wesentlichen die Ergebnisse der Gittersuche. Für die Populationsgröße wurde in beiden Fällen die maximale Anzahl von 100 Individuen gewählt. Während bei der Gittersuche der Wert noch auf $\{1, 50, 100\}$ beschränkt war, konnte die modellbasierte Optimierung den Wert im Intervall $[1, 100]$ frei wählen. Dass alle zehn besten Konfigurationen die obere Intervallgrenze von 100 Individuen nutzen, legt nahe, dass das tatsächliche Optimum noch höher liegen könnte. Dies widerspricht in Teilen den Erkenntnissen aus der Literatur, wo kleinere Populationsgrößen gewählt wurden. Auffällig ist auch die Wahl der Crossoverwahrscheinlichkeit von 0%, was das Crossover vollständig aus dem Algorithmus entfernt. Sowohl die paarweisen Vergleiche nach der Gittersuche, als auch die besten Konfigurationen der modellbasierten Parameteroptimierung legen einen Verzicht auf Crossoveroperatoren zugunsten einer höheren Mutationszahl nahe. In der Literatur stellten sich Crossoveroperatoren hingegen als hilfreich heraus. Dies lässt sich möglicherweise durch die kleinen Instanzgrößen begründen. Bei diesen existieren tendenziell weniger lokale Minima, sodass eine gute Konvergenz durch eine hohe initiale Lösungsdiversität in Folge einer hohen Populationszahl in Kombination mit den Mutationsoperatoren erreicht werden kann. Crossoveroperatoren sorgen im Allgemeinen für den Erhalt einer Lösungsdiversität bei gleichzeitigem Erhalt der Lösungsgüte durch Kombination geeigneter Teile der Chromosomen. Durch die große Population und der beschränkten Zahl an Iterationsläufen ist eine solche Diversifikation nicht nötig und nutzt Funktionsauswertungen, die durch Mutationen sinnvoller eingesetzt werden können.

Die Mutationswahrscheinlichkeit wurde von der Gittersuche maximiert, bei einer Elternschaftsgröße von 0.55. Bei fehlendem Crossover ergibt sich die mittlere Anzahl Mutationen pro Iteration als $par_size \times p_m$, was im Fall der Gittersuche 55%, im Fall der modellbasierten Parameteroptimierung 36% der Individuen sind. Nach jeder Iteration wird eine Überlebensselektion durchgeführt, was durch den Selektionsdruck die Konvergenz der Population fördert. Hier wurde von beiden Verfahren die aggressivere *Truncation selection* gewählt, die einen stärkeren Selektionsdruck als die Turnierselektion aufweist. Auch bei der Elternselektion wurde das Selektionsverfahren mit stärkerem Selektionsdruck, die Turnierselektion, verwendet. Bei der Gittersuche war die Turnierselektion gemittelt über alle Läufe noch signifikant schlechter als die zufällige Wahl, der Unterschied war aber bereits gering. Die besten beiden ermittelten Konfigurationen verwendeten bereits die Turnierselektion. Bei den Ergebnissen der modellbasierten Parameteroptimierung führt die Turnierselektion klar gegenüber der zufälligen Wahl.

Entgegen der Annahme, dass das Modellfitting der modellbasierten Parameteroptimierung gegenüber der Laufzeit des Algorithmus zu vernachlässigen sei, stellte sich in den Experimenten heraus, dass gegen Ende der 1.696 Läufe die Berechnungsdauer signifikant wurde. Vorteil des Surrogatmodells soll die schnelle Prädiktion neuer Versuchspunkte und somit die eine Reduzierung zeitintensiver Versuchsläufe sein. Durch die hohe Zahl der bereits ausgewerteten Punkte wird das Modellfitting laufend teurer und das Verfahren somit ineffizienter. Die beste Konfiguration wurde jedoch zu diesem Zeitpunkt bereits ermittelt, sodass dies keinen Einfluss auf die Qualität der Auswertung hatte. Zusammen mit den 2.592 Läufen der Gittersuche resultiert ein Modell mit 4.288 Versuchspunkten. Ansatz zur Reduktion der Berechnungsdauer des Surrogatmodells kann die Verwerfung von Versuchspunkten ab einer gewählten Obergrenze sein. Das verwendete Framework *skopt* stellt dazu eine automatische Bestimmung nicht-signifikanter Versuchspunkte zur Verfügung.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Vorgehensmodell zur automatischen Konfiguration von Algorithmen entwickelt. Das Konzept baut auf den Erkenntnissen aus der vorangegangenen Bachelorarbeit und den Arbeiten der Projektgruppe Autoschedule zur Komponentisierung von Algorithmen und ihrer automatischen Komposition mittels CLS auf. Es wurde gezeigt, dass CLS nicht nur zur Synthese aller Inhabitanten der Grammatik verwendet werden kann, sondern auch zur Komposition einzelner Lösungen einsetzbar ist. Dies ermöglicht die gezielte Konfiguration von Algorithmenvarianten zur Parameteroptimierung. Durch Hinzunahme des Luigi-Frameworks ist es zusätzlich möglich, numerische Parameter zu übergeben. An das Syntheseframework wurden zwei Parameteroptimierer angebunden, eine Gittersuche und eine modellbasierte Parameteroptimierung. Die Optimierung der Parameter eines Algorithmus ist von essenzieller Bedeutung für deren Performanz auf verschiedenen Problemen. Es ist daher von Vorteil, die Optimierung automatisiert durchführen zu können. Die Variationspunkte werden unter Zuhilfenahme von CLS automatisch aus einer gegebenen Menge an Kombinatoren ermittelt, sodass zur Generierung des Parametervektors keine weitere Konfiguration nötig ist. Auch die numerischen Parameter können automatisch erkannt und mittels Luigi konfiguriert werden. Derzeit wird die Manipulation des Repositorys derart realisiert, dass genau die eine gewollte Konfiguration generiert wird. Dies ist durch das Zusammenspiel von CLS und Luigi nötig, da im aktuellen Konzept nicht zwei Algorithmen mit verschiedenen numerischen Parametern synthetisiert werden können. Dies stellt einen Ansatz zur Verbesserung der Performanz des Frameworks, insbesondere bei der Gittersuche mit großen Überschneidungen zwischen den gewählten Parametern dar. Hierzu muss die Synthese der numerischen Parameter modularisiert werden. Für die modellbasierte Parameteroptimierung mit geringen Überscheidungen zwischen den Parametern und insbesondere der sequentiellen Auswertung von derzeit nur einer Konfiguration, stellt dies hingegen keine Veränderung dar.

Beide Parameteroptimierer wurden zur Ermittlung einer guten Konfiguration für den Evolutionären Algorithmus für ein gegebenes Benchmarkproblem aus der Literatur erfolgreich angewendet. Zur Einordnung der Lösungsgüte wurde ein mathematisches Optimierungsmodell für die betrachtete Problemklasse entworfen und für die Probleminstanzen ausgeführt. Gemessen an der RPD konnte bereits die Gittersuche eine gute Konfiguration für den Evolutionären Algorithmus berechnen. Die anschließende Nachoptimierung mittels der modellbasierten Parameteroptimierung bestätigte in weiten Teilen die Erkenntnisse aus der Gittersuche und generierte eine Algorithmenkonfiguration, die für alle Testinstanzen annähernd optimale Ergebnisse lieferte. Die Übertragbarkeit der resultierenden Parameterwerte auf weitere Testinstanzen wurde bislang nicht untersucht. Es wurde gezeigt, dass die Parameter für verschiedene Problemklassen stark schwanken, eine Folgefrage kann daher die Übertragung der Parametrisierung auf ein Problem mit derselben Problemcharakte-

ristik sein. Zu untersuchen ist hierfür jedoch zunächst, welche Charakteristik überhaupt einen Einfluss auf die Parametrisierung hat. Es ist nicht klar, wie „identische Problemcharakteristik“ mit Sicherheit zu definieren ist. Durch Variation einzelner Nebenbedingungen oder Verteilungen für die Eingabewerte lassen sich gegebenenfalls Einflussfaktoren bestimmen. Dies kann helfen, Einschätzungen zu treffen, wann eine gegebene Heuristik auch auf andere Probleme anwendbar ist, und wann ein solcher Charakteristikwechsel potenziell zu einer Verschlechterung der Algorithmenperformanz ohne erneutes Tuning der Parameter führt. Es ist des Weiteren nicht klar, inwieweit die berechneten Algorithmenkonfigurationen bereits auf die gewählten Daten überangepasst sind.

Die Parameteroptimierung wurde zunächst nur auf den entwickelten Evolutionären Algorithmus angewendet. In den Vorarbeiten wurden jedoch bereits andere Verfahren wie Lokale Suchen implementiert, die ebenfalls Variationspunkte und numerische Parameter beinhalten. Die Einbindung dieser Implementierungen in das Repository würde es ermöglichen, die verschiedenen Verfahren mit optimierten Parametern zu vergleichen. Bislang wurden diese Vergleiche ausschließlich mit wenigen Parameterwerten durchgeführt, da die händische Konfiguration dieser Werte aufwändig und daher in der Menge der Verfahren nicht zielführend ist. In der Literaturanalyse wurde gezeigt, dass evolutionäre Ansätze häufig als die besseren Verfahren, verglichen mit anderen Heuristiken, genannt werden. Unter Zuhilfenahme von CLS lassen sich eine deutlich größere Auswahl von Verfahrensvarianten synthetisieren, sodass ein Vergleich einer großen Bandbreite an Lösungsstrategien ermöglicht wird, um diese Erkenntnisse zu bestätigen oder potenziell neue und bessere Verfahren zu generieren. Durch die Komponentisierung verschiedener Lösungsstrategien lassen sich Konzepte zwischen diesen automatisiert übertragen und somit neue Kombinationen erzeugen und testen, die bislang nicht untersucht wurden.

Literatur

[1] Jatinder N. D. Gupta. „Two-Stage, Hybrid Flowshop Scheduling Problem“. In: *The Journal of the Operational Research Society* 39.4 (Apr. 1988), S. 359–364. ISSN: 01605682. DOI: 10.2307/2582115.

[2] Dominik Mäckel. *Synthese von Scheduling-Heuristiken für Flow Shop- und Job Shop-Probleme zur Makespanminimierung durch Komponentisierung und Rekombination.* Bachelorarbeit. Unveröffentlichtes Manuskript. Dortmund: Technische Universität Dortmund, 2020.

[3] Dominik Mäckel, Jan Winkels und Christin Schumacher. „Synthesis of Scheduling Heuristics by Composition and Recombination“. In: *Optimization and Learning.* Hrsg. von Bernabé Dorronsoro, Lionel Amodeo, Mario Pavone und Patricia Ruiz. Bd. 1443. Cham: Springer International Publishing, 2021, S. 283–293. ISBN: 978-3-030-85671-7. DOI: 10.1007/978-3-030-85672-4_21.

[4] Serhiy Danilevych, Marco Eckey, Nico Gorecki, Dominik Mäckel, Alexander Ostrop, Marcel Rohlf, Marc Schneider und Christoph Stockhoff. *Automatische Generierung von Maschinenbelegungsalgorithmen für die Fertigung eines Automobilzulieferers.* Unveröffentlichtes Manuskript. Dortmund: Technische Universität Dortmund, 2021.

[5] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens und Jakob Rehof. „Combinatory Logic Synthesizer“. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change.* Hrsg. von Tiziana Margaria und Bernhard Steffen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, S. 26–40. ISBN: 978-3-662-45234-9. DOI: 10.1007/978-3-662-45234-9_3.

[6] Agoston E. Eiben und Selmar Kagiso Smit. „Parameter tuning for configuring and analyzing evolutionary algorithms“. In: *Swarm and Evolutionary Computation* 1.1 (1. März 2011), S. 19–31. ISSN: 2210-6502. DOI: 10.1016/j.swevo.2011.02.001.

[7] Florian Jaehn und Erwin Pesch. *Ablaufplanung: Einführung in Scheduling.* Berlin, Heidelberg: Springer, 2014. ISBN: 978-3-642-54438-5. DOI: 10.1007/978-3-642-54439-2.

[8] Jan Herrmann. *Supply Chain Scheduling.* Wiesbaden: Gabler, 2010. ISBN: 978-3-8349-2266-3. DOI: 10.1007/978-3-8349-8667-2.

[9] Michael Pinedo. *Scheduling: theory, algorithms, and systems.* 5. Aufl. Cham Heidelberg New York Dordrecht London: Springer, 2016. ISBN: 978-3-319-79973-5. DOI: 10.1007/978-3-319-26580-3.

[10] Günther Schuh und Volker Stich, Hrsg. *Produktionsplanung und -steuerung 1: Evolution der PPS.* Berlin, Heidelberg: Springer, 2012. ISBN: 978-3-642-25422-2. DOI: 10.1007/978-3-642-25423-9.

[11] Robert J. Wittrock. „An Adaptable Scheduling Algorithm for Flexible Flow Lines". In: *Operations Research* 36.3 (1988), S. 445–453. ISSN: 0030-364X. DOI: 10.1287/opre.36.3.445.

[12] Ronald L. Graham, Eugene Leighton Lawler, Jan Karel Lenstra und A. H. G. Rinnooy Kan. „Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey". In: *Annals of Discrete Mathematics*. Hrsg. von P. L. Hammer, E. L. Johnson und B. H. Korte. Bd. 5. Discrete Optimization II. Elsevier, 1. Jan. 1979, S. 287–326. DOI: 10.1016/S0167-5060(08)70356-X.

[13] Rubén Ruiz, Funda Sivrikaya Şerifoğlu und Thijs Urlings. „Modeling realistic hybrid flexible flowshop scheduling problems". In: *Computers & Operations Research* 35.4 (1. Apr. 2008), S. 1151–1175. ISSN: 0305-0548. DOI: 10.1016/j.cor.2006.07.014.

[14] Thijs Urlings. „Heuristics and metaheuristics for heavily constrained hybrid flowshop problems". Diss. Valencia (Spain): Universitat Politècnica de València, 29. Juni 2010. DOI: 10.4995/Thesis/10251/8439.

[15] Alain Pétrowski und Sana Ben-Hamida. *Evolutionary Algorithms*. 1. Aufl. Wiley, 2. Mai 2017. ISBN: 978-1-84821-804-8. DOI: 10.1002/9781119136378.

[16] Agoston E. Eiben und Jim E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Berlin, Heidelberg: Springer, 2015. ISBN: 978-3-662-44873-1. DOI: 10.1007/978-3-662-44874-8.

[17] Kenneth Sörensen und Fred W. Glover. „Metaheuristics". In: *Encyclopedia of Operations Research and Management Science*. Hrsg. von Saul I. Gass und Michael C. Fu. Boston, MA: Springer US, 2013, S. 960–970. ISBN: 978-1-4419-1153-7. DOI: 10.1007/978-1-4419-1153-7_1167.

[18] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata, 15. Stuttgart-Bad Cannstatt: Frommann-Holzboog, 1973. 170 S. ISBN: 978-3-7728-0373-4.

[19] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Reading, Mass: Addison-Wesley Pub. Co, 1989. 412 S. ISBN: 978-0-201-15767-3.

[20] Patrick Siarry, Hrsg. *Metaheuristics*. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-45401-6. DOI: 10.1007/978-3-319-45403-0.

[21] Kenneth Sörensen, Marc Sevaux und Fred Glover. „A history of metaheuristics". In: *Handbook of Heuristics*. 13. Aug. 2018, S. 791–808. DOI: 10.1007/978-3-319-07124-4_4.

[22] Yongsheng Fang und Jun Li. „A Review of Tournament Selection in Genetic Programming". In: *Advances in Computation and Intelligence*. Hrsg. von Zhihua Cai, Chengyu Hu, Zhuo Kang und Yong Liu. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, S. 181–192. ISBN: 978-3-642-16493-4. DOI: 10.1007/978-3-642-16493-4_19.

[23] Huayang Xie, Mengjie Zhang und Peter Andreae. „Another investigation on tournament selection: Modelling and visualisation“. In: Proceedings of GECCO 2007: Genetic and Evolutionary Computation Conference. London, UK, 7. Juli 2007, S. 1468–1475. DOI: 10.1145/1276958.1277226.

[24] Lawrence Davis, Hrsg. *Handbook of genetic algorithms*. New York: Van Nostrand Reinhold, 1991. 385 S. ISBN: 978-0-442-00173-5.

[25] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown und Kevin P. Murphy. „An experimental investigation of model-based parameter optimisation: SPO and beyond“. In: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. GECCO ’09. New York, USA: Association for Computing Machinery, 8. Juli 2009, S. 271–278. ISBN: 978-1-60558-325-9. DOI: 10.1145/1569901.1569940.

[26] Raji Ghawi und Jürgen Pfeffer. „Efficient Hyperparameter Tuning with Grid Search for Text Categorization using kNN Approach with BM25 Similarity“. In: *Open Computer Science* 9.1 (1. Jan. 2019), S. 160–180. ISSN: 2299-1093. DOI: 10.1515/comp-2019-0011.

[27] Messias Borges Silva, Hrsg. *Design of Experiments - Applications*. InTech, 26. Juni 2013. ISBN: 978-953-51-1168-9. DOI: 10.5772/45728.

[28] Patricia Y. Kuo, Hai Du, Lloyd Andrew Corkan, Kexin Yang und Jonathan S. Lindsey. „A planning module for performing grid search, factorial design, and related combinatorial studies on an automated chemistry workstation“. In: *Chemometrics and Intelligent Laboratory Systems* 48.2 (2. Aug. 1999), S. 219–234. ISSN: 0169-7439. DOI: 10.1016/S0169-7439(99)00021-0.

[29] Gerhard Krennrich. *4 Design of Experiments (DoE) | Experimental Design and Process Optimization with R*. 10. Feb. 2020.

[30] Mauro Birattari, Thomas Stützle, Luis Paquete und Klaus Varrentrapp. „A Racing Algorithm for Configuring Metaheuristics.“ In: Proceedings of the Genetic and Evolutionary Computation Conference. New York, USA, 1. Jan. 2002, S. 11–18.

[31] Frank Hutter, Holger H. Hoos und Kevin Leyton-Brown. „Sequential Model-Based Optimization for General Algorithm Configuration“. In: *Learning and Intelligent Optimization*. Hrsg. von Carlos A. Coello Coello. Bd. 6683. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 507–523. ISBN: 978-3-642-25565-6. DOI: 10.1007/978-3-642-25566-3_40.

[32] William H. Press, Hrsg. *Numerical recipes: the art of scientific computing*. 3rd ed. Cambridge, UK; New York, USA: Cambridge University Press, 2007. 1235 S. ISBN: 978-0-521-88068-8.

[33] Roman Garnett. *Bayesian optimization*. Cambridge, United Kingdom ; New York, NY: Cambridge University Press, 2023. ISBN: 978-1-108-42578-0.

[34] Jonas Mockus. „The Bayesian Approach to Local Optimization“. In: *Bayesian Approach to Global Optimization: Theory and Applications*. Hrsg. von Jonas Mockus. Mathematics and Its Applications. Dordrecht: Springer Netherlands, 1989, S. 125–156. ISBN: 978-94-009-0909-0. DOI: `10.1007/978-94-009-0909-0_7`.

[35] Jasper Snoek, Hugo Larochelle und Ryan P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms*. 29. Aug. 2012. DOI: `10.48550/arXiv.1206.2944`. arXiv: `1206.2944[cs,stat]`.

[36] The scikit-optimize contributors. *scikit-optimize: sequential model-based optimization in Python — scikit-optimize 0.8.1 documentation*. Version 0.9.0. 12. Okt. 2021. URL: `https://scikit-optimize.github.io/stable/` (besucht am 14. 08. 2023).

[37] David Cornapeau, Olivier Grise, Varoquaux Gaël, Alexandre Gramfort und Andreas Mueller. *scikit-learn: machine learning in Python — scikit-learn 1.3.0 documentation*. Version 1.3.0. 30. Juni 2023. URL: `https://scikit-learn.org/stable/` (besucht am 14. 08. 2023).

[38] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, John C. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vanderplas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt und SciPy 1.0 Contributors. „SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python“. In: *Nature Methods* 17 (2020), S. 261–272. DOI: `10.1038/s41592-019-0686-2`.

[39] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke und Travis E. Oliphant. „Array programming with NumPy“. In: *Nature* 585 (2020), S. 357–362. DOI: `10.1038/s41586-020-2649-2`.

[40] Jakob Rehof und Pawel Urzyczyn. *Finite Combinatory Logic with Intersection Types*. Technical Reports In Computer Science 834. Dortmund: Technische Universität Dortmund. Fakultät für Informatik, Feb. 2011.

[41] J. Roger Hindley und Jonathan P. Seldin. *Introduction to combinators and [lambda]-calculus*. London Mathematical Society student texts 1. Cambridge, UK; New York, USA: Cambridge University Press, 1986. ISBN: 978-0-521-26896-7.

[42] Kaizhou Gao, Zhiguang Cao, Le Zhang, Zhenghua Chen, Yuyan Han und Quanke Pan. „A review on swarm intelligence and evolutionary algorithms for solving flexible job shop scheduling problems“. In: *IEEE/CAA Journal of Automatica Sinica* 6.4 (Juli 2019), S. 904–916. ISSN: 2329-9274. DOI: 10.1109/JAS.2019.1911540.

[43] Rubén Ruiz und José Antonio Vázquez-Rodríguez. „The hybrid flow shop scheduling problem“. In: *European Journal of Operational Research* 205.1 (16. Aug. 2010), S. 1–18. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2009.09.024.

[44] Christin Schumacher. *Anpassungsfähige Maschinenbelegungsplanung eines praxisorientierten hybriden Flow Shops*. Wiesbaden: Springer Fachmedien, 2023. ISBN: 978-3-658-41169-5. DOI: 10.1007/978-3-658-41170-1.

[45] Mas Gholami, Mostafa Zandieh und Akbar Alem-Tabriz. „Scheduling hybrid flow shop with sequence-dependent setup times and machines with random breakdowns“. In: *The International Journal of Advanced Manufacturing Technology* 42.1 (1. Mai 2009), S. 189–201. ISSN: 1433-3015. DOI: 10.1007/s00170-008-1577-3.

[46] Amy D. Wilson, Russel E. King und Thom J. Hodgson. „Scheduling non-similar groups on a flow line: multiple group setups“. In: *Robotics and Computer-Integrated Manufacturing*. 13th International Conference on Flexible Automation and Intelligent Manufacturing 20.6 (1. Dez. 2004), S. 505–515. ISSN: 0736-5845. DOI: 10.1016/j.rcim.2004.07.002.

[47] Victor Yaurima, Larisa Burtseva und Andrei Tchernykh. „Hybrid flowshop with unrelated machines, sequence-dependent setup time, availability constraints and limited buffers“. In: *Computers & Industrial Engineering* 56.4 (1. Mai 2009), S. 1452–1463. ISSN: 0360-8352. DOI: 10.1016/j.cie.2008.09.004.

[48] Rubén Ruiz und Concepción Maroto. „A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility“. In: *European Journal of Operational Research* 169.3 (16. März 2006), S. 781–800. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2004.06.038.

[49] Mostafa Zandieh, E. Mozaffari und Mohammad Gholami. „A robust genetic algorithm for scheduling realistic hybrid flexible flow line problems“. In: *Journal of Intelligent Manufacturing* 21.6 (1. Dez. 2010), S. 731–743. ISSN: 1572-8145. DOI: 10.1007/s10845-009-0250-5.

[50] Fantahun Melaku Defersha und Mingyuan Chen. „Mathematical model and parallel genetic algorithm for hybrid flexible flowshop lot streaming problem“. In: *The International Journal of Advanced Manufacturing Technology* 62.1 (1. Sep. 2012), S. 249–265. ISSN: 1433-3015. DOI: 10.1007/s00170-011-3798-0.

[51] Seyedeh Sarah Zabihzadeh und Javad Rezaeian. „Two meta-heuristic algorithms for flexible flow shop scheduling problem with robotic transportation and release time“. In: *Applied Soft Computing* 40 (1. März 2016), S. 319–330. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2015.11.008.

[52] Binh Minh Nguyen, Huynh Thi Thanh Binh, Tran The Anh und Do Bao Son. „Evolutionary Algorithms to Optimize Task Scheduling Problem for the IoT Based Bag-of-Tasks Application in Cloud–Fog Computing Environment“. In: *Applied Sciences* 9.9 (Jan. 2019), S. 1730–1749. ISSN: 2076-3417. DOI: 10.3390/app9091730.

[53] Razieh Saremi, Hardik Yagnik, Julian Togelius, Ye Yang und Guenther Ruhe. *An Evolutionary Algorithm for Task Scheduling in Crowdsourced Software Development*. 5. Juli 2021. DOI: 10.48550/arXiv.2107.02202. arXiv: 2107.02202[cs].

[54] Kalayanmoy Deb, Amrit Pratap, Sameer Agarwal und T. Meyarivan. „A fast and elitist multiobjective genetic algorithm: NSGA-II“. In: *IEEE Transactions on Evolutionary Computation* 6.2 (Apr. 2002), S. 182–197. ISSN: 1941-0026. DOI: 10.1109/4235.996017.

[55] Fuqing Zhao, Xuan He und Ling Wang. „A Two-Stage Cooperative Evolutionary Algorithm With Problem-Specific Knowledge for Energy-Efficient Scheduling of No-Wait Flow-Shop Problem“. In: *IEEE Transactions on Cybernetics* 51.11 (Nov. 2021), S. 5291–5303. ISSN: 2168-2275. DOI: 10.1109/TCYB.2020.3025662.

[56] Setareh Majidi. „Evolutionary algorithm for scheduling real-time applications in system of systems“. Doctoral Thesis. Siegen: Universität Siegen, 2022. DOI: 10.25819/ubsi/10141.

[57] Joachim Wegener, Harmen Sthamer, Bryan F. Jones und David E. Eyres. „Testing real-time systems using genetic algorithms“. In: *Software Quality Journal* 6.2 (1. Juni 1997), S. 127–135. ISSN: 1573-1367. DOI: 10.1023/A:1018551716639.

[58] Mohit Kumar Kakkar, Jajji Singla, Neha Garg, Gourav Gupta, Prateek Srivastava und Ajay Kumar. „Class Schedule Generation using Evolutionary Algorithms“. In: *Journal of Physics: Conference Series* 1950.1 (Aug. 2021), S. 012067. ISSN: 1742-6596. DOI: 10.1088/1742-6596/1950/1/012067.

[59] Jiaxin Fan, Yingli Li, Jin Xie, Chunjiang Zhang, Weiming Shen und Liang Gao. „A Hybrid Evolutionary Algorithm Using Two Solution Representations for Hybrid Flow-Shop Scheduling Problem“. In: *IEEE Transactions on Cybernetics* 53.3 (März 2023), S. 1752–1764. ISSN: 2168-2275. DOI: 10.1109/TCYB.2021.3120875.

[60] F. Jabbarizadeh, Mostafa Zandieh und Davoud Talebi. „Hybrid flexible flowshops with sequence-dependent setup times and machine availability constraints“. In: *Computers & Industrial Engineering* 57.3 (1. Okt. 2009), S. 949–957. ISSN: 0360-8352. DOI: 10.1016/j.cie.2009.03.012.

[61] Orhan Engin, Gülşad Ceran und Mustafa K. Yilmaz. „An efficient genetic algorithm for hybrid flow shop scheduling with multiprocessor task problems“. In: *Applied Soft Computing* 11.3 (1. Apr. 2011), S. 3056–3065. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2010.12.006.

[62] Jitti Jungwattanakit, Manop Reodecha, Paveena Chaovalitwongse und Frank Werner. „Algorithms for flexible flow shop problems with unrelated parallel machines, setup times, and dual criteria“. In: *The International Journal of Advanced Manufacturing Technology* 37.3 (1. Mai 2008), S. 354–370. ISSN: 1433-3015. DOI: 10.1007/s00170-007-0977-0.

[63] Boris Naujoks. „Design and tuning of an evolutionary multiobjective optimisation algorithm“. Diss. Dortmund: Technische Universität Dortmund, 6. Apr. 2011. DOI: 10.17877/DE290R-13412.

[64] Thomas Bartz-Beielstein, Christian W. G. Lasarczyk und Mike Preuss. „Sequential parameter optimization“. In: *2005 IEEE Congress on Evolutionary Computation.* 2005 IEEE Congress on Evolutionary Computation. Bd. 1. ISSN: 1941-0026. Sep. 2005, 773–780 Vol.1. DOI: 10.1109/CEC.2005.1554761.

[65] Thomas Bartz-Beielstein, Christian Lasarczyk und Mike Preuß. *SPOT Sequential Parameter Optimization Toolbox.* Dortmund: Technische Universität Dortmund, Dez. 2008. DOI: 10.17877/DE290R-9033.

[66] Thomas Bartz-Beielstein und Sandor Markon. *Tuning search algorithms for real-world applications.* Dortmund: Technische Universität Dortmund, 13. Juli 2004. DOI: 10.17877/DE290R-15365.

[67] Lisa Lenz, Julian Graefenstein, Jan Winkels und Mike Gralla. „Smart Factory Adaption Planning by means of BIM in Combination of Constraint Solving Techniques“. In: CIB World Building Congress 2019. Hongkong, 17. Juni 2019.

[68] Jan Winkels. „Automatisierte Komposition und Konfiguration von Workflows zur Planung mittels kombinatorischer Logik“. Diss. Dormund: Technische Universität Dortmund, 2019. DOI: 10.17877/DE290R-20469.

[69] Julian Graefenstein, Jan Winkels, Lisa Lenz, Kai Christian Weist, Kevin Krebil und Mike Gralla. „A Hybrid Approach of Modular Planning – Synchronizing Factory and Building Planning by Using Component based Synthesis“. In: Hawaii International Conference on System Sciences. Hawaii, 1. Jan. 2020. DOI: 10.24251/HICSS.2020.806.

[70] Fadil Kallat, Tristan Schäfer und Anna Vasileva. „CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories“. In: *Electronic Proceedings in Theoretical Computer Science* 301 (23. Aug. 2019), S. 51–65. ISSN: 2075-2180. DOI: 10.4204/EPTCS.301.7. arXiv: 1908.09481[cs].

[71] Alexander Mages, Carina Mieth, Jens Hetzler, Fadil Kallat, Jakob Rehof, Christian Riest und Tristan Schafer. „Automatic Component-Based Synthesis of User-Configured Manufacturing Simulation Models“. In: Winter Simulation Conference. Singapur, 11. Dez. 2022, S. 1841–1852. DOI: `10.1109/WSC57314.2022.10015425`.

[72] Fadil Kallat, Jakob Pfrommer, Jan Bessai, Jakob Rehof und Anne Meyer. „Automatic Building of a Repository for Component-based Synthesis of Warehouse Simulation Models“. In: *Procedia CIRP*. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0 104 (1. Jan. 2021), S. 1440–1445. ISSN: 2212-8271. DOI: `10.1016/j.procir.2021.11.243`.

[73] Fadil Kallat, Carina Mieth, Jakob Rehof und Anne Meyer. „Using Component-based Software Synthesis and Constraint Solving to generate Sets of Manufacturing Simulation Models“. In: *Procedia CIRP*. 53rd CIRP Conference on Manufacturing Systems 2020 93 (1. Jan. 2020), S. 556–561. ISSN: 2212-8271. DOI: `10.1016/j.procir.2020.03.018`.

[74] Fadil Kallat. „Komponentenbasierte Synthese von Simulationsmodellen“. Diss. Dortmund: Technische Universität Dortmund, 2022. DOI: `10.17877/DE290R-23750`.

[75] Tristan Schäfer, Jan Bessai, Constantin Chaumet, Jakob Rehof und Christian Riest. „Design Space Exploration for Sampling-Based Motion Planning Programs with Combinatory Logic Synthesis“. In: *Algorithmic Foundations of Robotics XV*. Hrsg. von Steven M. LaValle, Jason M. O’Kane, Michael Otte, Dorsa Sadigh und Pratap Tokekar. Springer Proceedings in Advanced Robotics. Cham: Springer International Publishing, 2023, S. 36–51. ISBN: 978-3-031-21090-7. DOI: `10.1007/978-3-031-21090-7_3`.

[76] Tristan Schäfer. „Component-based synthesis of motion planning algorithms“. Diss. Dortmund: Technische Universität Dortmund, 2021. DOI: `10.17877/DE290R-22992`.

[77] Erik Bernhardsson und Elias Freider. *Luigi*. 14. Sep. 2023. URL: `https://github.com/spotify/luigi` (besucht am 14. 09. 2023).

[78] Anne Meyer, Jan Bessai, Hadi Kutabi und Daniel Scholtyssek. „CLS-Luigi: A Framework for Automated Synthesis and Execution of Decision Pipelines“. In: LION 17. Nizza, Frankreich, 2023.

[79] Benedikt Julian Kordus. *Anpassung sowie Evaluation von konstruktiven Heuristiken und Standarddatensätzen für hybride Flow Shops der Maschinenbelegungsplanung*. Bachelorarbeit. Unveröffentlichtes Manuskript. Dortmund: Technische Universität Dortmund, 2022.

[80] Emmanuel Néron, Philippe Baptiste und Jatinder N. D. Gupta. „Solving hybrid flow shop problem using energetic reasoning and global operations“. In: *Omega* 29.6 (1. Dez. 2001), S. 501–511. ISSN: 0305-0483. DOI: `10.1016/S0305-0483(01)00040-8`.

[81] Bahman Naderi, Rubén Ruiz und Mostafa Zandieh. „Algorithms for a realistic variant of flowshop scheduling“. In: *Computers & Operations Research* 37.2 (1. Feb. 2010), S. 236–246. ISSN: 0305-0548. DOI: 10.1016/j.cor.2009.04.017.

[82] Quan-Ke Pan, Rubén Ruiz und Pedro Alfaro-Fernández. „Iterated search methods for earliness and tardiness minimization in hybrid flowshops with due windows“. In: *Computers & Operations Research* 80 (1. Nov. 2016). DOI: 10.1016/j.cor.2016.11.022.

[83] Sebastian Lang, Tobias Reggelin, Fabian Behrendt und Abdulrahman Nahhas. „Evolving Neural Networks to Solve a Two-Stage Hybrid Flow Shop Scheduling Problem with Family Setup Times“. In: The 53rd Hawaii International Conference on System Sciences. Hawaii, 9. Jan. 2020. DOI: 10.24251/HICSS.2020.160.

[84] Ahmed Missaoui und Rubén Ruiz. „A parameter-Less iterated greedy method for the hybrid flowshop scheduling problem with setup times and due date windows“. In: *European Journal of Operational Research* 303.1 (16. Nov. 2022), S. 99–113. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2022.02.019.

[85] Christoph Laroque, Madlene Leißau, Pedro Copado Méndez, Christin Schumacher, Javier Panadero und Angel Juan. „A Biased-Randomized Discrete-Event Algorithm for the Hybrid Flow Shop Problem with Time Dependencies and Priority Constraints“. In: *Algorithms* 15(2):54 (1. Feb. 2022). DOI: 10.3390/a15020054.

[86] James Rumbaugh, Ivar Jacobson und Grady Booch. *The unified modeling language reference manual.* 2nd ed. The Addison-Wesley object technology series. Boston: Addison-Wesley, 2005. 721 S. ISBN: 978-0-321-24562-5.

[87] Guido van Rossum, Barry Warsaw und Nick Coghlan. *Style Guide for {Python} Code.* PEP 8. 2001.

[88] *importlib — The implementation of import.* Version 3.11.4. 27. Juli 2023. URL: https://docs.python.org/3/library/importlib.html (besucht am 28. 07. 2023).

[89] Karlheinz Zwerenz. „Statistik: Einführung in die computergestützte Datenanalyse“. In: *Statistik.* Oldenbourg Wissenschaftsverlag, 18. Sep. 2012. ISBN: 978-3-486-71899-7. DOI: 10.1524/9783486718997.

[90] Roxy Peck, Chris Olsen und Jay L. Devore. *Introduction to statistics and data analysis.* 3rd ed. Australia ; Belmont, CA: Thomson Brooks/Cole, 2008. 847 S. ISBN: 978-0-495-11873-2.

[91] Drew Fralick, Julia Z. Zheng, Bokai Wang, Xin M. Tu und Changyong Feng. „The Differences and Similarities Between Two-Sample T-Test and Paired T-Test“. In: *Shanghai Archives of Psychiatry* 29.3 (2017), S. 184–188. ISSN: 1002-0829. DOI: 10.11919/j.issn.1002-0829.217070.

[92] Amanda Ross und Victor L. Willson. „Paired Samples T-Test“. In: *Basic and Advanced Statistical Tests: Writing Results Sections and Creating Tables and Figures.* Hrsg. von Amanda Ross und Victor L. Willson. Rotterdam: SensePublishers, 2017, S. 17–19. ISBN: 978-94-6351-086-8. DOI: 10.1007/978-94-6351-086-8_4.

Für diese Arbeit wurden Beratungsleistungen zur Auswahl geeigneter statistischer Analyse- und Darstellungsmethoden des Bereichs „Statistische Beratung und Analyse“ (SBAZ) des Zentrums für Hochschulbildung der TU Dortmund in Anspruch genommen.

Anhang

Schedule
+ machine_arrangement : np.ndarray + num_stages : int + num_machines : int + num_jobs : int + p_ilj : np.ndarray + unavail_il : np.ndarray = None + skip_ij : np.ndarray = None + e_ilj : np.ndarray = None + r_j : np.ndarray = None + rm_il : np.ndarray = None + d_j : np.ndarray = None + w_j : np.ndarray = None + prec_j : np.ndarray = None + s_iljk : np.ndarray = None + a_iljk : np.ndarray = None + lag_ilj : np.ndarray = None + batch_il : np.ndarray = None + prmu : bool = False + due_dates_as_deadlines : bool = False − __arr : np.ndarray
[...]

Abbildung 26: Darstellung aller Felder des Schedule-Objekts in UML.

Schedule

[...]

+ ___init___(self, machine_arrangement : np.ndarray, p_ilj : np.ndarray, [...])
+ ___iter___(self) : Generator[tuple[int, int, int], Schedule, None]
+ allocate(self, stage : int, machine : int, job : int, start_time : int) : None
+ allocate_backwards(self, stage : int, machine : int, job : int, end_time : int) : None
+ allocate_tight(self, stage : int, machine : int, job : int) : None
+ sort_allocations(self) : None
+ get_allocations(self, stage : int, machine : int) : list[Allocation]
+ update_objective(self, objective : object, value : float) : None
+ get_unique_jobs(self) : list[int]
+ get_jobs_on_stage(self, stage : int = 0) : list[int]
+ get_processing_time_on_machine(self, stage : int, machine : int, job : int) : int
+ get_avg_processing_time_on_stage(self, stage : int, job : int) : float
+ get_scheduled_jobs(self) : list[tuple[int, int, int, int]]
+ get_scheduled_jobs_on_stage(self, stage : int = 0) : list[tuple[int, int, int]]
+ get_scheduled_jobs_on_stages(self, stages : set[int]) : list[tuple[int, int, int]]
+ get_machines_on_stage(self, stage : int) : int
+ get_machine_finish_time(self, stage : int, machine : int) : int
+ get_machine_finish_times_on_stage(self, stage : int) : list[int]
+ get_all_machine_finish_times(self) : list[int]
+ get_eligible_machines(self, stage : int, job : int) : list[tuple[int, int]]
+ clear_stage(self, stage : int) : None
+ clear_stages(self, start_stage : int) : None
+ clear_machine(self, stage : int, machine : int) : None
+ copy(self) : Schedule
+ shallow_copy(self) : Schedule
+ equals(other : Schedule) : bool
+ pretty_print(self, show_time=False) : None

Abbildung 27: Darstellung aller öffentlicher Methoden des Schedule-Objekts in UML.

Algorithmus 23 Rekombinationsoperator Order-Based.

Require: Eltern s_1, s_2.

```
1: s ← s_1.shallow_copy()
2: machines ← s.get_machines_on_stage(stage = 0)
3: machine = random(machines)
4: π_1 ← s_1.get_allocations(stage = 0, machine = machine)        ▷ Hole Chromosomen
5: π_2 ← s_2.get_allocations(stage = 0, machine = machine)
6: if |π_1| == 0 ∨ |π_2| == 0 then
7:     return s
8: end if
9: r_1 ← random(0, |π_1|)                                         ▷ Bestimme Crossoverpunkte
10: r_2 ← random(0, |π_1|)
11: c_1 ← min(r_1, r_2)
12: c_2 ← max(r_1, r_2)
13: π ← [ ]                                  ▷ Kopiere Gene aus π_1 zwischen c_1 und c_2
14: for i ∈ [c_1, c_2 − 1] do
15:     π[i] ← π_1[i]
16: end for
17: k ← c_2 mod |π_1|                                   ▷ Kopiere Gene aus π_2 nach c_2
18: for Auftrag job ∈ π_2 do
19:     if job ∈ π_1 ∧ job ∉ π then
20:         π[k] ← job
21:         k ← (k + 1) mod |π_1|
22:     end if
23: end for
24: for Auftrag job ∈ π_1 do                            ▷ Kopiere restliche Gene aus π_1
25:     if job ∉ π then
26:         π[k] ← job
27:         k ← (k + 1) mod |π_1|
28:     end if
29: end for
30: s.clear_machine(stage = 0, machine = machine)       ▷ Plane neues Chromosom ein
31: for Auftrag job ∈ π do
32:     s.allocate_tight(stage = 0, machine = machine, job = job)
33: end for
34: return s
```

Algorithmus 24 Rekombinationsoperator Maschinenzuweisungen.

Require: Eltern s_1, s_2.

```
1:  s ← s1.shallow_copy()
2:  machines = s1.get_machines_on_stage(stage = 0)
3:  c ← random(machines)                                   ▷ Bestimme Crossoverpunkt
4:  planned ← ∅
5:  for Maschine m ∈ [0, c − 1] do                         ▷ Kopiere Zuweisungen aus s1 bis c
6:      for Auftrag j ∈ s1.get_allocations(stage = 0, machine = m) do
7:          s.allocate_tight(stage = 0, machine = m, job = j)
8:          planned ← planned ∪ {j}
9:      end for
10: end for
11: for Maschine m ∈ [c, machines − 1] do                  ▷ Kopiere Zuweisungen aus s2 ab c
12:     for Auftrag j ∈ s2.get_allocations(stage = 0, machine = m) do
13:         if j ∉ planned then
14:             s.allocate_tight(stage = 0, machine = m, job = j)
15:             planned ← planned ∪ {j}
16:         end if
17:     end for
18: end for
19: for Maschine m ∈ [c, machines − 1] do                  ▷ Kopiere restliche Zuweisungen aus s1
20:     for Auftrag j ∈ s1.get_allocations(stage = 0, machine = m) do
21:         if j ∉ planned then
22:             s.allocate_tight(stage = 0, machine = m, job = j)
23:             planned ← planned ∪ {j}
24:         end if
25:     end for
26: end for
27: for Maschine m ∈ [0, c − 1] do                         ▷ Kopiere restliche Zuweisungen aus s2
28:     for Auftrag j ∈ s2.get_allocations(stage = 0, machine = m) do
29:         if j ∉ planned then
30:             s.allocate_tight(stage = 0, machine = m, job = j)
31:         end if
32:     end for
33: end for
34: return s
```

Algorithmus 25 Mutationsoperator Shift.

Require: Schedule s.

```
1: jobs ← s.get_scheduled_jobs_on_stage(stage = 0)
2: if |jobs| == 0 then
3:     return s
4: end if
5: ▷ Wähle zufälligen Auftrag und neue Maschine
6: j, m_1, index_1 ← random(jobs).
7: m_2, p_{0 m_2 j} ← random(schedule.get_eligible_machines(stage = 0, job = j))
8: ▷ Hole Allokationsobjekte
9: allocations_1 ← s.get_allocations(stage = 0, machine = m_1)
10: allocations_2 ← s.get_allocations(stage = 0, machine = m_2)
11: ▷ Ziehe Position auf neuer Maschine
12: num_jobs ← len(allocations_2)
13: if m_1 == m_2 then
14:     num_jobs ← num_jobs − 1
15: end if
16: index_2 ← random(0, num_jobs)
17: ▷ Entferne Auftrag von ursprünglicher Maschine
18: allocation ← allocations_1.pop(index_1)
19: for alloc ∈ allocatons_1[index_1 :] do
20:     alloc.move_start_time(delta = −allocation.processing_time)
21: end for
22: ▷ Füge Auftrag neuer Position auf neuer Maschine ein
23: allocations_2.insert(index_2, allocation)
24: new_start_time ← 0
25: if index_2 > 0 then
26:     new_start_time ← allocations_2[index_2 − 1].end_time
27: end if
28: allocation.set_time(start_time = new_start_time, processing_time = p_{0 m_2 j})
29: ▷ Verschiebe spätere Aufträge
30: for alloc ∈ allocatons_2[index_2 + 1 :] do
31:     alloc.move_start_time(delta = p_{0 m_2 j})
32: end for
33: return s
```

Algorithmus 26 Mutationsoperator Swap.

Require: Schedule s.

1: $jobs \leftarrow s.get_scheduled_jobs_on_stage(stage = 0)$
2: **if** $|jobs| \leq 1$ **then**
3: **return** s
4: **end if**
5: $jobs \leftarrow shuffle(jobs)$
6: $j_1, m_1, index_1 \leftarrow jobs[0]$
7: **for** $j_2, m_2, index_2 \in jobs[1:]$ **do**
8: ▷ Prüfe gegenseitige Maschinenqualifikation
9: **if** $\neg e_{0 m_1 j_2} \vee \neg e_{0 m_2 j_1}$ **then**
10: **continue**
11: **end if**
12: ▷ Hole Allokationsobjekte
13: $allocations_1 \leftarrow s.get_allocations(stage = 0, machine = m_1)$
14: $allocations_2 \leftarrow s.get_allocations(stage = 0, machine = m_2)$
15: $allocation_1 \leftarrow allocations_1[index_1]$
16: $allocation_2 \leftarrow allocations_2[index_2]$
17: ▷ Errechne Zeitverschiebungen
18: $d_1 \leftarrow s.get_processing_time_on_machine(stage = 0, machine = m_1, job = j_2)$
19: $d_1 \leftarrow d_1 - allocation_1.processing_time$
20: $d_2 \leftarrow s.get_processing_time_on_machine(stage = 0, machine = m_2, job = j_1)$
21: $d_2 \leftarrow d_2 - allocation_2.processing_time$
22: ▷ Tausche Aufträge
23: $allocation_1.job \leftarrow j_2$
24: $allocation_1.alter_processing_time(delta = d_1)$
25: $allocation_2.job \leftarrow j_1$
26: $allocation_2.alter_processing_time(delta = d_2)$
27: ▷ Verschiebe spätere Aufträge
28: **for** $alloc \in allocations_1[index_1 + 1:]$ **do**
29: $allocation.move_start_time(delta = d_1)$
30: **end for**
31: **for** $alloc \in allocations_2[index_2 + 1:]$ **do**
32: $allocation.move_start_time(delta = d_2)$
33: **end for**
34: **return** s
35: **end for**
36: **return** s

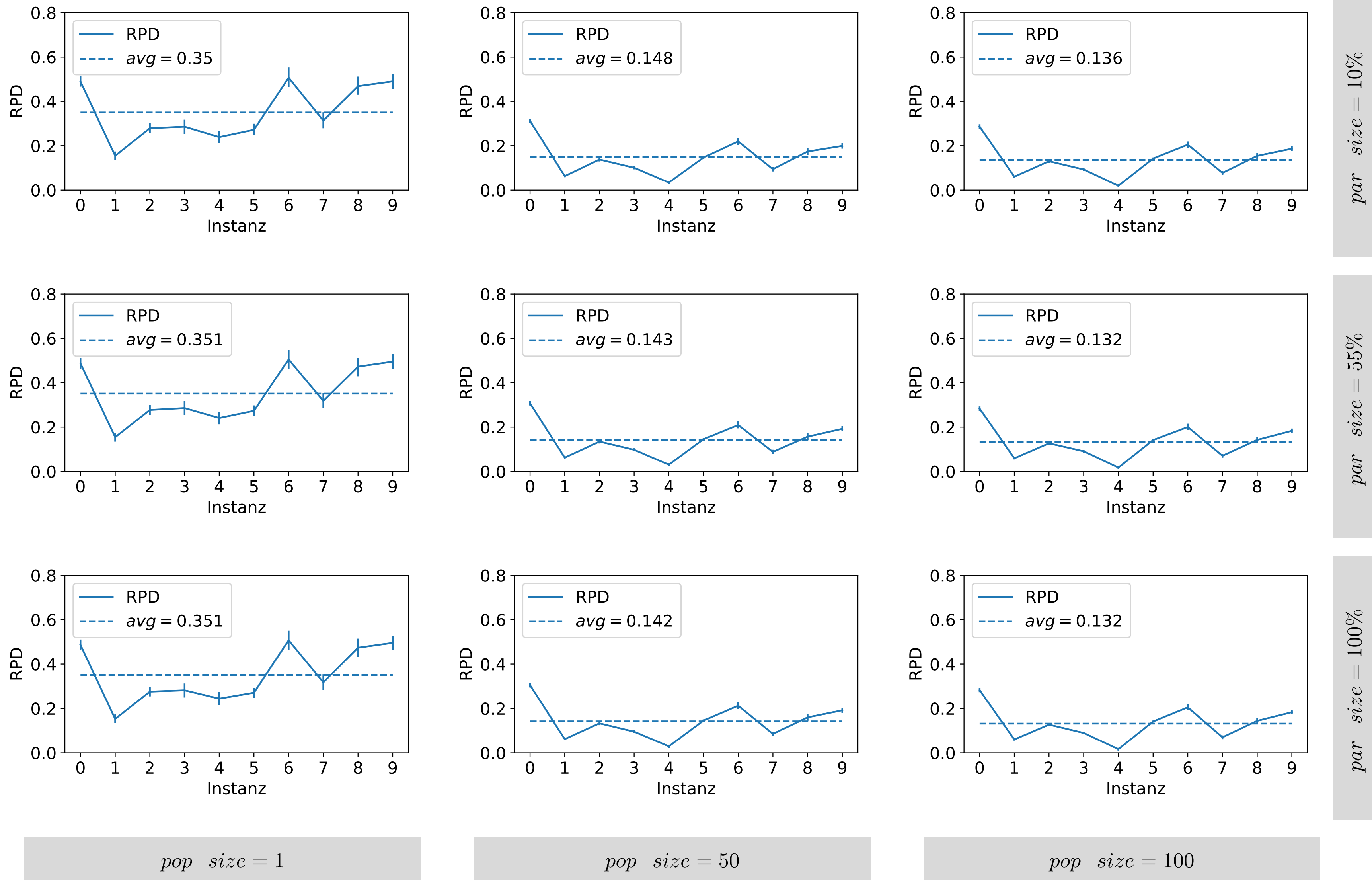


Abbildung 28: GG-Plot der Beziehungen zwischen Populationsgröße und Elternschaft.